**AFRL-IF-RS-TR-1999-163**
**Final Technical Report**
**August 1999**

# ROBUST LOCALLY OPTIMUM DETECTION IN AUTO-REGRESSIVE NOISE

**Illinois Institute of Technology**
**Department of Electrical and Computer Engineering**

Myron R. Mychal and Donald R. Ucci

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

AFRL-IF-RS-TR-1999-163 has been reviewed and is approved for publication.

APPROVED: [illegible]

JOHN J. PATTI
Project Engineer

FOR THE DIRECTOR: [illegible]

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE<br>August 1999 | 3. REPORT TYPE AND DATES COVERED<br>Final Jun 95 - Jun 98 |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>ROBUST LOCALLY OPTIMUM DETECTION IN AUTO-REGRESSIVE NOISE | 5. FUNDING NUMBERS<br>C - F30602-95-C-0215<br>PE - 62702F<br>PR - 4519<br>TA - 42<br>WU - 93 |
| 6. AUTHOR(S)<br>Myron R. Mychal and Donal R. Ucci | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>Illinois Institute of Technology<br>Department of Electrical and Computer Engineering<br>Chicago IL 60616-3793 | 8. PERFORMING ORGANIZATION REPORT NUMBER<br>N/A |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br>Air Force Research Laboratory/IFGC<br>525 Brooks Road<br>Rome NY 13441-4505 | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER<br>AFRL-IF-RS-TR-1999-163 |
| 11. SUPPLEMENTARY NOTES<br>Air Force Research Laboratory Project Engineer: John J. Patti/IFGC/(315) 330-3615 | |
| 12a. DISTRIBUTION AVAILABILITY STATEMENT<br>Approved for public release; distribution unlimited | 12b. DISTRIBUTION CODE |

13. ABSTRACT *(Maximum 200 words)*

This report presents the derivation and simulation of a robust locally optimum (LO) nonlinear spread spectrum receiver. The signaling environment consists of the desired received signal in correlated interference and thermal noise. Autoregressive (AR) spectral modeling methods and a histogram approximation of the probability density function are employed. The system under consideration is a binary phase-shift keyed (BPSK) communications system. Results from simulations indicate that applying AR modeling to the environment improves performance substantially in most cases. This document summarizes the work to date.

| | | | |
|---|---|---|---|
| 14. SUBJECT TERMS<br>Communications, Signal Processing, Signal Detection, Spread Spectrum | | | 15. NUMBER OF PAGES<br>68<br>16. PRICE CODE |
| 17. SECURITY CLASSIFICATION OF REPORT<br>UNCLASSIFIED | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>UNCLASSIFIED | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>UNCLASSIFIED | 20. LIMITATION OF ABSTRACT<br>UL |

## Contents

# 1 Introduction

Many facets are involved in the design of a spread spectrum communications system [23]. One important consideration is determining a method to best recover the transmitted signal when it is subjected to interference in the transmission path. This interference is often highly correlated and not necessarily Gaussian as in typical interference models [1]. The linear correlator realization of the matched filter is no longer optimal for this interference environment. *Locally optimum* (LO) detection provides a method for circumventing this problem if the *probability density function* (pdf) of the interference is known. However, in cases where the interference exhibits strong self-correlation, traditional LO methods exhibit poor performance [3].

LO detectors with memory more successfully combat this type of disturbance. The disadvantage of a memory-based processor is the rapid increase in the dimensionality of the *joint probability density function* (jpdf) noise vector making the LO detector nonlinearity unwieldy [2]. To alleviate this problem, frequency domain methods are used to determine a $P^{TH}$ *-order autoregressive* (AR(P)) model of the channel disturbance. The AR methodology reduces the dimensionality of the underlying jpdf to that of the model order. The statistics of the jpdf remain an issue. To this end, *probability density function* (pdf) estimation techniques for *independent identically distributed* (iid) noise samples prove useful (11) since the input sequence to the AR model is iid, albeit characterized by some unknown pdf.

The necessary AR model parameters must be estimated by well-known spectral estimation techniques [1]. Considering that each received symbol will be limited by the sampling rate, an estimation technique must provide accurate results for a finite number of observations of the random process. The Yule-Walker Method [1] works best when the number of samples approaches infinity. Other techniques [4] can estimate the model parameters given very few samples.

This report presents the derivation and simulation of a robust *locally optimum* (LO) nonlinear spread spectrum receiver. The signaling environment consists of the desired received signal in correlated interference and thermal noise. Autoregressive (AR) spectral modeling methods and a histogram approximation of the probability density function are employed. The system under consideration is a *binary phase-shift keyed* (BPSK) communications system. Results from simulations [9] indicate that applying AR modeling to the environment improves performance substantially in most cases. This document summarizes the work to date.

## 2 The Locally Optimum Detector

Consider the case of binary detection of real, discrete signals in additive noise. In binary signal detection the goal is to decide which of two possible information signals is present at the detector. For example, in a *binary phased shift keyed* (BPSK) communications system [22], the receiver must decide whether a value of +1 or −1 was sent by the transmitter. Stated more formally, the goal of the detector is to correctly choose one of the following two hypotheses:

$$\begin{aligned} H_1 &: \text{Signal } \mathbf{s}_1 \text{ present} \qquad (1)\\ H_0 &: \text{Signal } \mathbf{s}_0 \text{ present,} \end{aligned}$$

where $\mathbf{s}_1$ and $\mathbf{s}_0$ are vectors and the notation $\mathbf{x} = [x_1 \cdots x_N]^T$ denotes a column vector of length $N$. The value of $N$ is the number of signal samples in a given observation period.

The derivation of the corresponding LO detector with memory is summarized by Jacklin and Ucci in Chapter 9 of [11], and is repeated here for convenience. To begin, let the received signal vector of length $N$ be given by

$$\mathbf{r} = \mathbf{s}_m + \mathbf{n}, \tag{2}$$

where $m = 0$ or $1$ and $\mathbf{n}$ is a random noise vector. Given the observation $\mathbf{r} = \boldsymbol{\rho}$, the optimum detector is of the form [17]

$$\ln[\lambda(\boldsymbol{\rho})] \underset{choose\ H_0}{\overset{choose\ H_1}{\gtrless}} \tilde{\gamma}, \tag{3}$$

where the log likelihood ratio is given by [17]

$$\ln[\lambda(\boldsymbol{\rho})] = \ln\left[\frac{f_{\mathbf{r}}(\boldsymbol{\rho} \mid H_1)}{f_{\mathbf{r}}(\boldsymbol{\rho} \mid H_0)}\right] = \ln\left[\frac{f_{\mathbf{n}}(\boldsymbol{\rho} - \mathbf{s}_1)}{f_{\mathbf{n}}(\boldsymbol{\rho} - \mathbf{s}_0)}\right], \tag{4}$$

assuming that $\mathbf{s}_m$ and $\mathbf{n}$ are independent. Approximating Eq. (4) using a first-order Taylor series expansion yields the following result:

$$\ln \lambda(\boldsymbol{\rho})] \approx -\sum_{i=1}^{N} (s_{1i} - s_{0i}) \frac{\partial}{\partial \rho_i} \ln[f_{\mathbf{n}}(\boldsymbol{\rho})]. \tag{5}$$

¿From Eq. (5), the resulting decision statistic becomes

$$l(\boldsymbol{\rho}) \triangleq \sum_{i=1}^{N} (s_{1i} - s_{0i})\, g_i(\boldsymbol{\rho}), \tag{6}$$

where

$$g_i(\boldsymbol{\rho}) \triangleq -\frac{\partial}{\partial \rho_i} \ln[f_{\mathbf{n}}(\boldsymbol{\rho})] = -\frac{\frac{\partial}{\partial \rho_i} f_{\mathbf{n}}(\boldsymbol{\rho})}{f_{\mathbf{n}}(\boldsymbol{\rho})} \tag{7}$$

is called the LO nonlinearity *with memory.* Thus, the LO detector with memory for the binary detection of known signals in additive noise is

$$l(\boldsymbol{\rho}) = \sum_{i=1}^{n} (s_{1i} - s_{0i})\, g_i(\boldsymbol{\rho}) \underset{choose\ H_0}{\overset{choose\ H_1}{\gtrless}} \tilde{\gamma}, \tag{8}$$

with $g_i(\boldsymbol{\rho})$ given by Eq. (7).

If $\mathbf{n}$ is a first-order Markov random sequence [24], then

$$f_{n_i}(\eta_i \mid \eta_{i-1}, \eta_{i-2}, \ldots, \eta_1) = f_{n_i}(\eta_i \mid \eta_{i-1}) . \tag{9}$$

Using Eq. (9), the joint pdf of $\mathbf{n}$ can be reduced to

$$f_{\mathbf{n}}(\boldsymbol{\eta}) = \prod_{i=1}^{N} f_{n_i}(\eta_i \mid \eta_{i-1}) , \tag{10}$$

where $f_{n_1}(\eta_1 \mid \eta_0) \stackrel{\triangle}{=} f_{n_1}(\eta_1)$. The LO nonlinearity requires $f_{\mathbf{n}}(\boldsymbol{\rho})$, the noise pdf evaluated at the value of the received signal. From Eq. (10) this is seen to be

$$f_{\mathbf{n}}(\boldsymbol{\rho}) = \prod_{i=1}^{N} f_{n_i}(\rho_i \mid \rho_{i-1}) . \tag{11}$$

Substituting Eq. (11) into Eq. (7), the LO nonlinearity with memory, $g_i(\boldsymbol{\rho}) = -\frac{\partial}{\partial \rho_i} \ln[f_{\mathbf{n}}(\boldsymbol{\rho})]$, becomes

$$g_i(\boldsymbol{\rho}) = \begin{cases} -\frac{\partial}{\partial \rho_i}\left(\ln[f_{n_i}(\rho_i \mid \rho_{i-1})] + \ln[f_{n_{i+1}}(\rho_{i+1} \mid \rho_i)]\right) , \\ \qquad\qquad\qquad\qquad \text{for } i = 1, \ldots, N-1 \\ \\ -\frac{\partial}{\partial \rho_N} \ln[f_{n_N}(\rho_N \mid \rho_{N-1})] , \qquad \text{for } i = N . \end{cases} \tag{12}$$

Thus, for first-order Markov noise, the original $N$-variate function $g_i(\boldsymbol{\rho})$ reduces to a function of at most three variables: $\rho_{i-1}$, $\rho_i$, and $\rho_{i+1}$.

## 3 The LO Detector for Auto-Regressive Noise

The focus of this effort is the development of robust LO detection algorithms for general AR noise environments. Suppose the noise sample, $n_i$, at a discrete time instant, $i$, can be modeled as a $P^{th}$-order Markov process. Then the autoregressive noise model is [3]

$$n_i = \begin{cases} \sum_{j=1}^{P} a_j n_{i-j} + w_i, & \text{for } i \,\epsilon\, [1,\, N], \\ 0, & \text{for } i \leq 0 \end{cases} \tag{13}$$

where the $\{a_i\}$ are the AR($P$) coefficients and $w_i$ represents an *independent identically distributed* (iid) random process and the vectors contain $N$ samples. In this case, the noise pdf is,

$$f_{\mathbf{n}}(\boldsymbol{\eta}) = f_{\mathbf{n}}(\eta_1, \cdots, \eta_N) = \prod_{i=1}^{N} f_{n_i}(\eta_i \mid \eta_{i-1}, \cdots, \eta_{i-P}) \tag{14}$$

where,

$$\begin{aligned} f_{n_i}(\eta_i \mid \eta_{i-1}, \cdots, \eta_{i-P}) &\triangleq f_{n_i}(\eta_i) & \text{for } i = 1, \\ f_{n_i}(\eta_i \mid \eta_{i-1}, \cdots, \eta_{i-P}) &\triangleq f_{n_i}(\eta_i \mid \eta_{i-1}, \cdots, \eta_1) & \text{for } i = 2, \cdots, P \end{aligned} \tag{15}$$

If a "block approximation" [1] [1] is assumed, then $f_{\mathbf{n}}(\eta)$ can be written as,

$$f_{\mathbf{n}}(\boldsymbol{\eta}) = \prod_{i=1}^{N} f_w \left( -\sum_{j=0}^{P} a_j \eta_{i-j} \right) \tag{16}$$

where $f_w(\omega)$ is the pdf of the white noise process and $a_0 \triangleq -1$. For this case, the form of the nonlinearity is,

$$g_i(\boldsymbol{\nu}) \triangleq \sum_{l=0}^{\min(P, N-i)} a_l h' \left( -\sum_{j=0}^{P} a_j \nu_{i+l-j} \right) \text{ for } i \,\epsilon\, [1,\, N], \tag{17}$$

where,

$$\nu_k = \begin{cases} 0, & \text{for } k \leq 0, \\ \rho_k & \text{for } k \,\epsilon\, [1,\, N], \end{cases} \tag{18}$$

and $h'(\omega)$ is the derivative of $h(\omega)$, and

[1]In general, $\eta_1$ depends on the last "$P$" previous samples, which are assumed to be zero so that, $\eta_1 = w_1$. This only affects a small fraction of the terms for $N \gg P$.

$$h(\omega) \triangleq \ln\left[f_w(\omega)\right] \tag{19}$$

Now we have reduced the nonlinearity to Eq. (17) providing an intuitive understanding to the ensuing calculation. Eq. (17) is also quite suitable for a vector representation since the argument of $h'(x)$ can be represented as a dot product between the AR parameters and a received vector. In turn, $h'(x)$ is itself a matrix of these dot products multiplied by the vector of AR coefficients thus enabling the computation of the necessary nonlinearity for the symbol of interest.

## 4 Histogram and AR Estimation Techniques

Essential to the computation of the LO detector nonlinearity are the noise pdf, $f_w(\omega)$, and the autoregressive coefficients, $\{a_i\}$. Since neither the source statistics of the driving white noise, nor the AR coefficients are known *a priori*, they must be estimated.

The estimation of the pdf is performed via a histogram method employed in the LO detector without memory [1]. Pure noise data is not available, so the received data must be used. Then, a three-point derivative is used to compute $h'(\rho)$. The histogram approach is chosen because of its ease and simplicity of implementation and generally acceptable performance.

The AR coefficients are estimated using the *modified covariance algorithm* (MCA) [12] as developed [4] and simulated and tested earlier [5]. Recall that the MCA minimizes the sum of the forward and backward prediction errors:

$$\varepsilon_p^M = \varepsilon_p^+ + \varepsilon_p^- \tag{20}$$

where the forward prediction error is:

$$e_p^+(n) = x(n) + \sum_{k=1}^{p} a_p(k)x(n-k) \tag{21}$$

and likewise the backward prediction error is:

$$e_p^-(n) = x(n-p) + \sum_{k=1}^{p} a_p^*(k)x(n-p+k) \tag{22}$$

and the error is given by:

$$\varepsilon_p^M = \sum_{n=p}^{N} [|e_p^+(n)|^2 + |e_p^-(n)|^2] \tag{23}$$

This minimization of error leads to the normal equations that are central to the algorithm.

$$\sum_{k=1}^{p} [r_x(l,k) + r_x(p-k, p-l)]a_p(k) = -[r_x(l,0) + r_x(p, p-l)];\, l = 1, \ldots, p \tag{24}$$

where:

$$r_x(l,k) = \sum_{n=p}^{N} x(n-k)x^*(n-l) \tag{25}$$

and the $\{a_p\}$ vector gives the estimates of the AR model parameters as required. This is a non-windowing method similar to the covariance method [12], but it differs from the latter in that it minimizes the sum of the squares of the forward and backward predictor errors. The MCA is chosen since, when compared to other AR estimation methods, it often provides:

- stable, high resolution spectral estimates for a small number of samples
- exhibits lower sensitivity to phase and decreased peak shifting, and
- is not subject to spectral line splitting.

The Burg Method gives the benefit of providing the calculation without requiring the estimation of the autocorrelation of the sequence of interest and can give results similar to the Modified Covariance Algorithm. The Burg Method would be preferable in an embedded system with a finite amount of memory, such as a digital signal processor.

## 5 System Code Translation

To make the computation of the nonlinearity viable and efficient in an embedded system, a translation to C or C++ is an essential step toward real-time integration on a *digital signal processor* (DSP). Two possible implementations were pursued: one using ANSI C and the other using C++ [7]. In the translation to C, the core modules of filtering and histogram generation were successfully completed. The C++ translation presents a simulation of a typical DSSS communication system for the variety of detectors considered in Section 6.

Currently, the simulation system exists only in MATLAB© format. To port it to C/C++ a translation of individual MATLAB modules is necessary. The first module translated was based on MATLAB's *filter()* function which implements an *infinite impulse response* (IIR) filter [22]. We implemented the *direct form II structure* because of its simplicity, minimal memory requirements and fast performance. This structure requires $M + N + 1$ multiplications and the number of delays is of the order, $N$, where $M$ and $N$ are the number of zeros and poles, respectively, of the filter of interest.

The *filter()* function takes three null-terminated arrays as inputs: $b[\ ]$, representing the numerator coefficients, $a[\ ]$, representing the denominator coefficients, and $data_in[\ ]$ representing the input data vector. The output is a single vector, $y[\ ]$.

The next translated function was the MATLAB *hist()* function. For our purposes, the histogram function must perform the following tasks: return the distribution and bin centers of an input array among a specified number of bins.

Once implemented, both functions were tested and compared to results from a MATLAB simulation. In testing *filter.c*, a wide range of varying inputs and filter coefficients were examined. Integer and fractional input values (positive and negative) were tested as well as varying vector lengths. A filter that was implemented along with its input sequence are given as:

- $b[\,] = \{1.22, -2, -2.01, -4.87\}$
- $a[\,] = \{4, 2, 3, -1.54, 5.98, -0.55\}$
- $data\_in[\,] = \{3, 9.2, 5.11, 6, -6, 12, 0.77, 0.95, -3.3344\}$

Using MATLAB's *filter()* function as a baseline, the results of passing the above input signal through *filter.c* revealed virtually perfect agreement [7].

In testing *histogram.c*, a wide range of bin numbers and data values were examined. Similar to the *filter.c* case, fractional and integer data were used. To illustrate the performance of the C implementation, the following input signal was passed through the histogram function which was divided into $bin\_info = 5$ bins:

$$signal[\,] = \{10000, 85, 0, 0, 5, -66.7, 88, 90, 0.555, 3.66, -9.999, 4.665, -25.6\}$$

When compared with the results of the MATLAB simulation, again, virtually perfect agreement was found. The results were:

$bin\_centers = \{939.97, 2953.31, 4966.65, 6979.99, 8993.33\}$
$frequency\_count = \{12, 0, 0, 0, 1\}$

The C++ simulation implemented the full DSSS communication system. C++ routines were written to parallel the MATLAB simulation for the detectors analyzed in Section 6. An object-oriented approach to system design was utilized. This allows for a higher degree of flexibility [21]. The class hierarchies for the data structures were developed [8].

The base class for all of the signals is *Matrix*. The class, *Signal*, is a special case of the *Matrix* class with some additions. The *Signal* class consists of the *BPSK Signal*, the *Noise Signal* and the *Interferer Signal*. The *Noise Signal* represents the background thermal noise process and the *Interferer Signal* represents one of a library of different types of high-power interferers. Currently, the only two interference types are partial band and continuous wave.

The translation used *Template* classes allowing the simulation to be performed in both "float" and "double" data types. This allows an examination to be made of the trade-off between precision and efficiency.

Simulations were executed for the linear correlator for ISR's of 2dB, 5dB, 10dB, 20dB, and 25dB. The error performance observed was as expected i.e. as interferer power decreased, so did the probability of bit error. This paralleled the MATLAB simulation results. Run times of the entire C++ simulation were much faster than the MATLAB code. For a partial band interferer, the relative performances of the five detectors also matched the MATLAB simulation results.

## 6 Simulation Results

The following figures indicate the performance of the ARLO detector versus four other detectors: a linear correlator, the LO detector, a linear correlator and an LO detector both preceded by an AR whitening filter. The interference environment consists of of any one of the following: a partial band interferer, a CW tone, multiple CW tones or a mixture of partial band and CW interference. The total ISR for each case is 30dB with a processing gain of 12dB. The signal to thermal noise ratio is varied from as low as -10dB to +5dB. For a block diagram of a typical system, refer to [1] with the ARLO detector taking the place of the LO detector. Also included are typical spectra for the signals of interest.

In general, the ARLO detector provides robust performance in all interference types. It outperforms all other detectors in all cases with the exception of the single CW case, where it closely matches the performance of the LO detector without memory. In the partial band interference scenario, the ARLO detector shows drastic improvement over the LO detector especially.

## 7 Conclusions and Future Work

The derivation of Eq. (17) provides a framework for simulation of the autoregressive locally optimum nonlinearity in complex interference environments. Previous simulation [5] focused on low order models to test the validity of the estimation techniques. As can be seen from the simulation results in Section 6, the performance improvement carries over into more complex interference scenarios. As anticipated [3], the computational complexity of the nonlinearity is significantly reduced from an $N$-dimensional jpdf estimation to a $P$-dimensional one.

The material provided in this final report represents the results of the work performed during the current research effort. Given these results, the following issues have been identified as possible tasks to be performed in future research efforts:

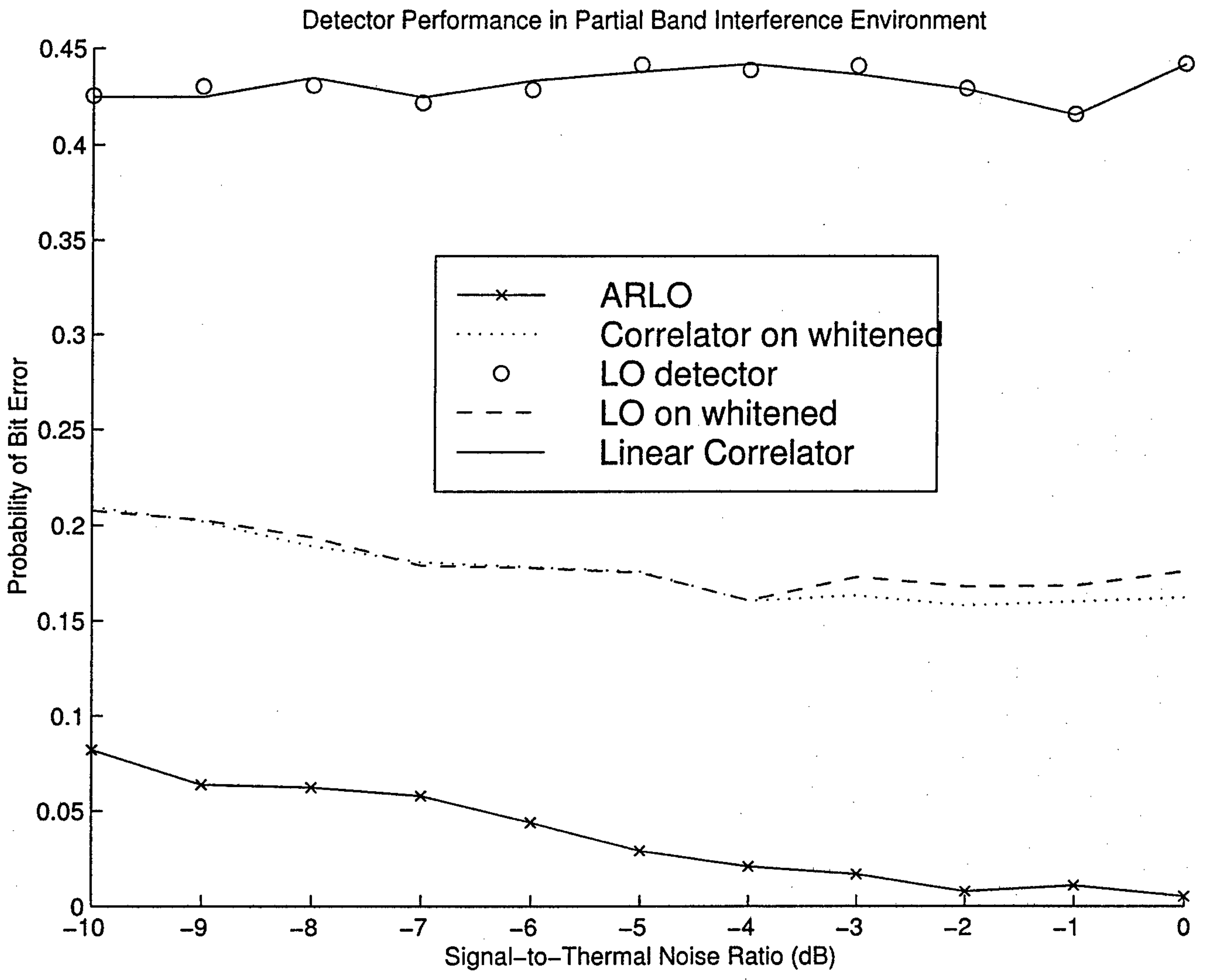


Figure 1: Partial Band environment for ISR = 30dB with PG = 12dB

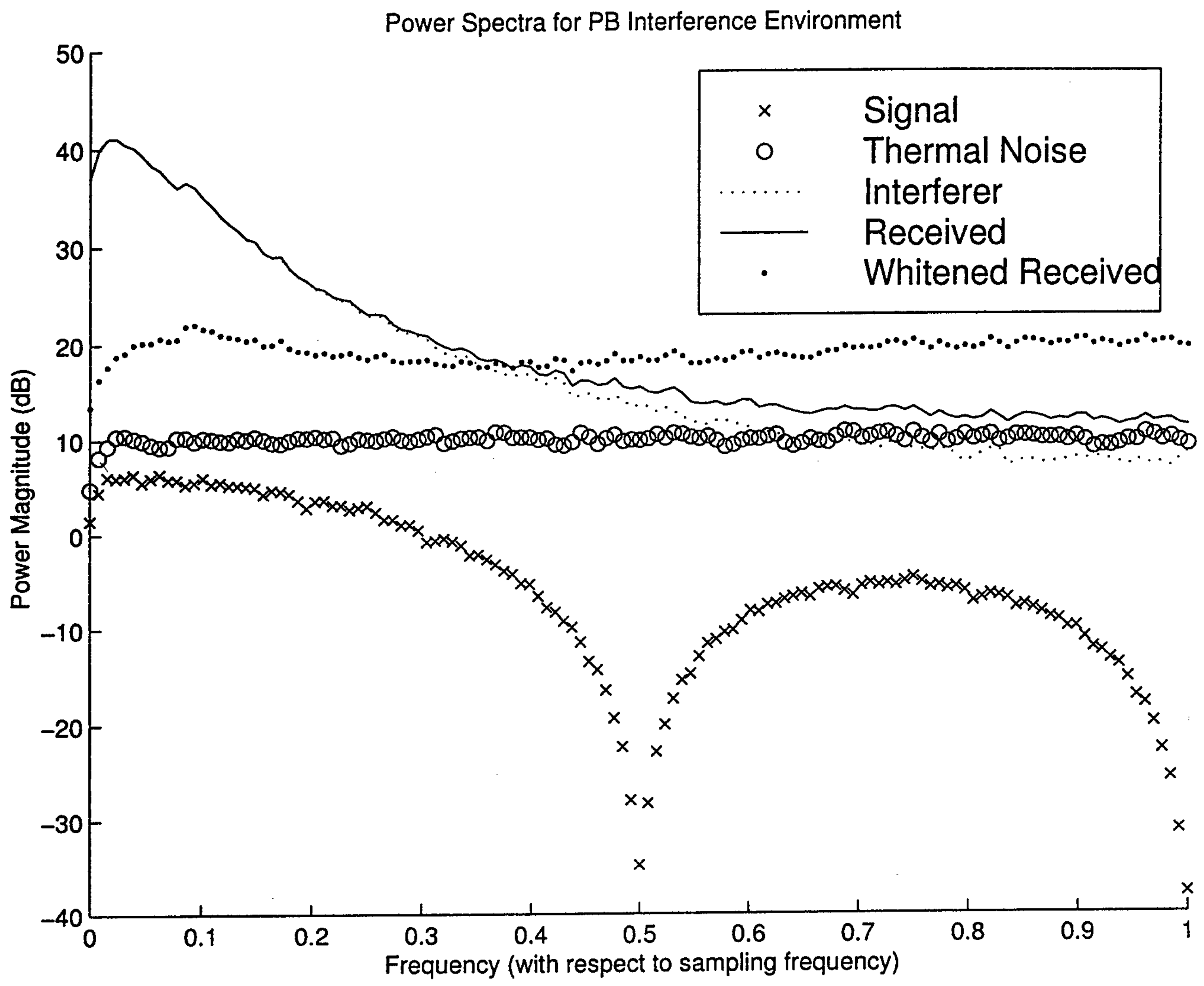


Figure 2: Partial Band spectra for ISR = 30dB with PG = 12dB

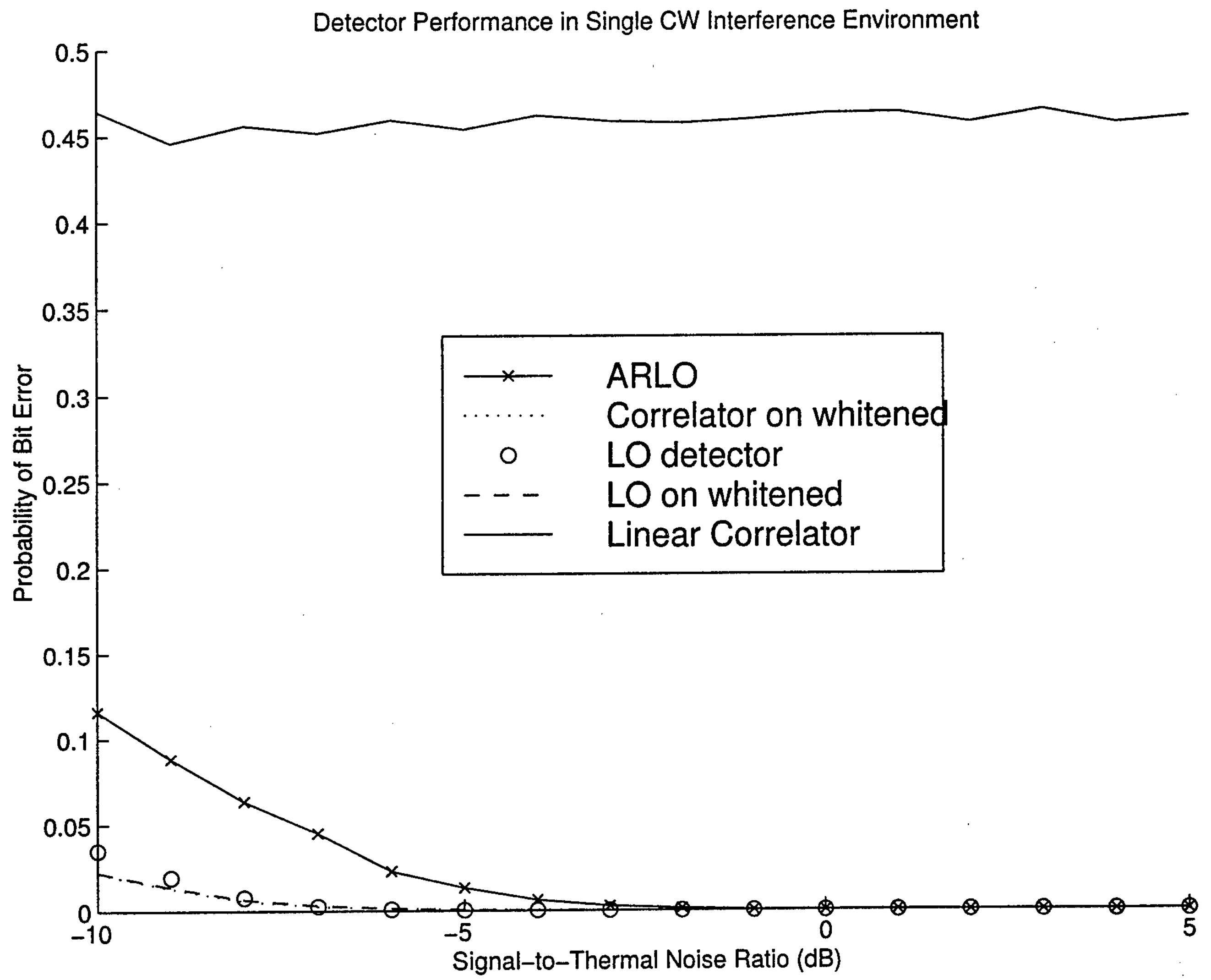


Figure 3: CW environment for the case of ISR = 30dB with PG = 12dB

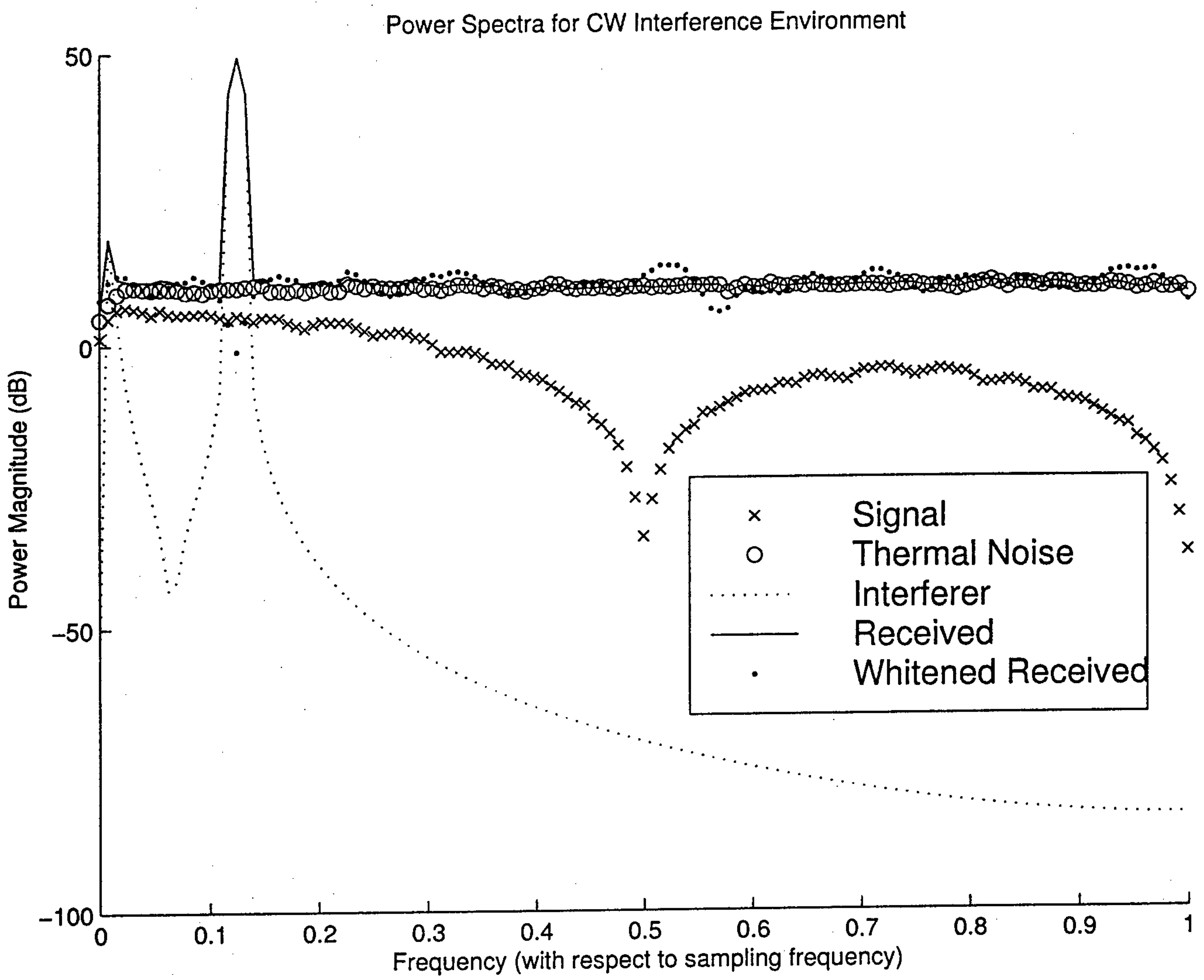


Figure 4: CW spectra for ISR = 30dB with PG = 12dB

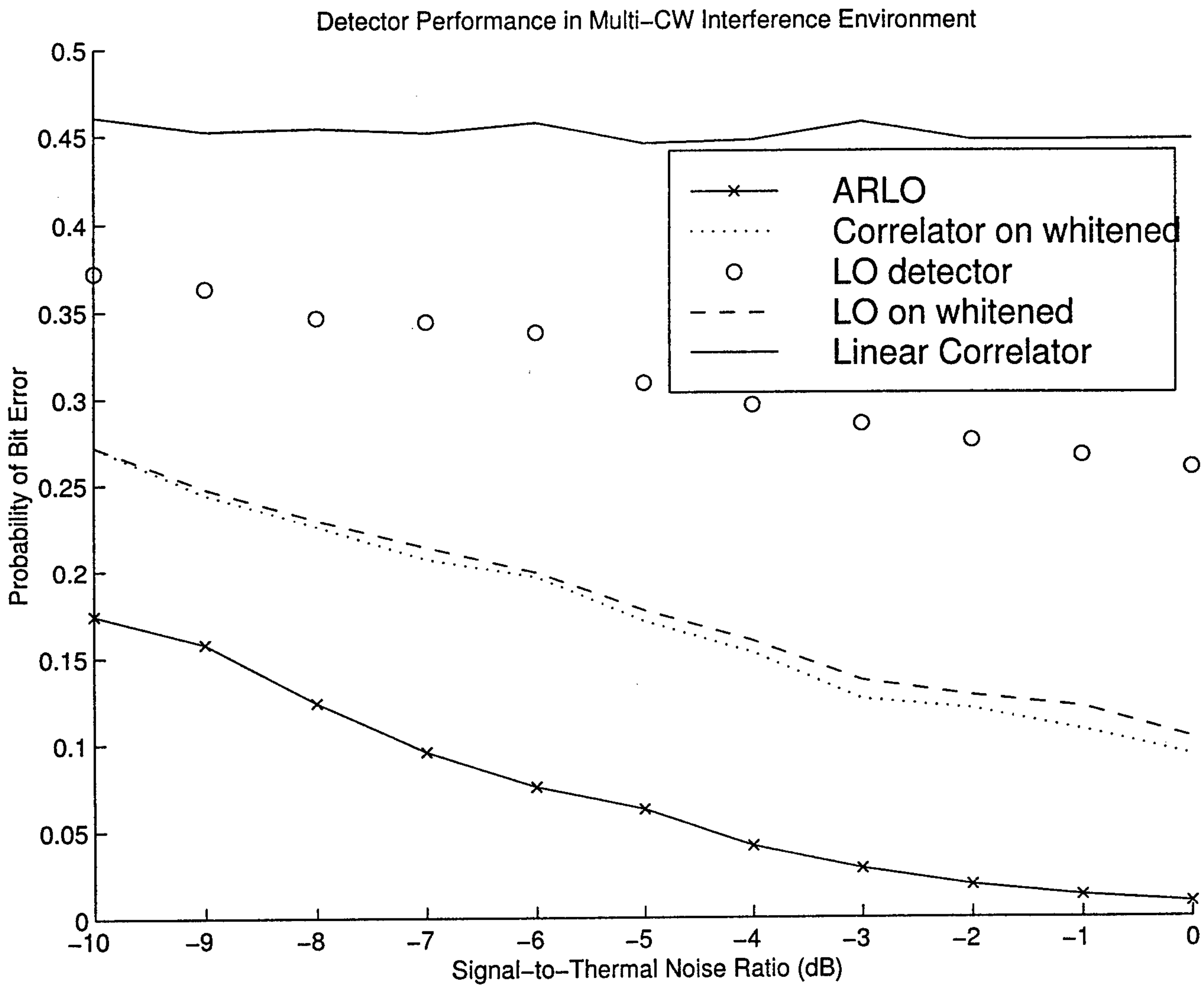


Figure 5: Multi-CW environment for total ISR = 30dB with PG = 12dB

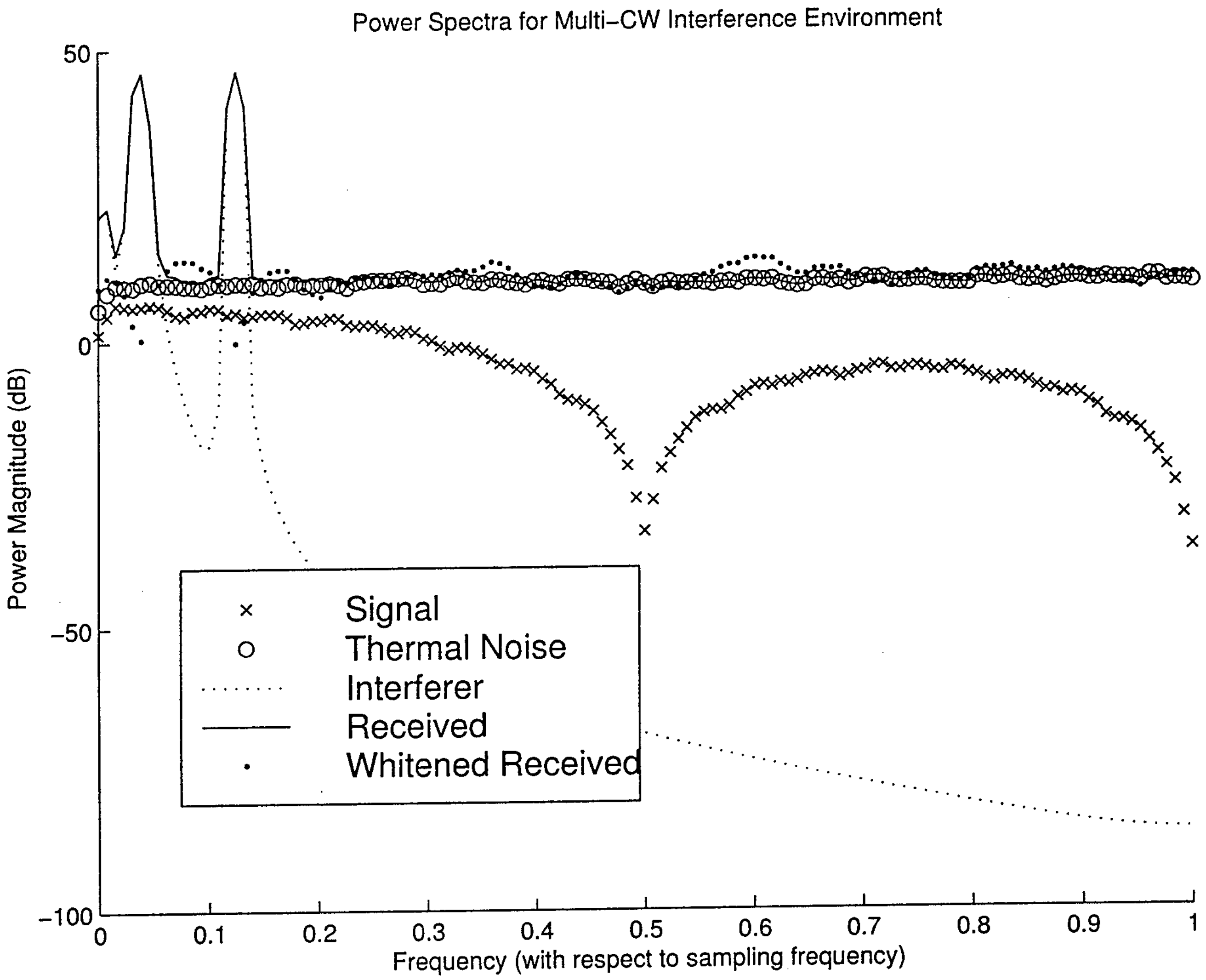


Figure 6: Multi-CW spectra for total ISR = 30dB with PG = 12dB

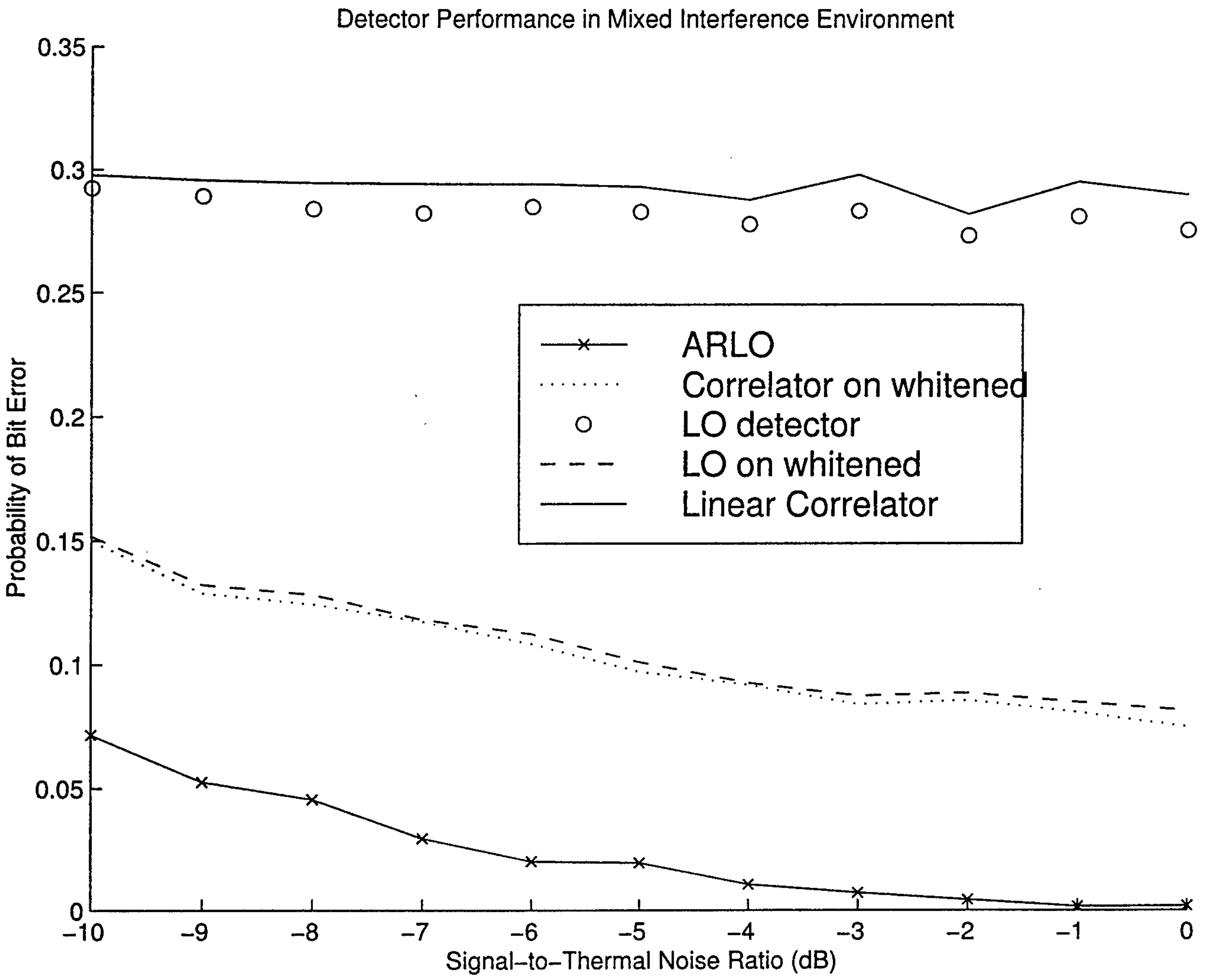


Figure 7: Mixed environment for ISR = 30dB with PG = 12dB

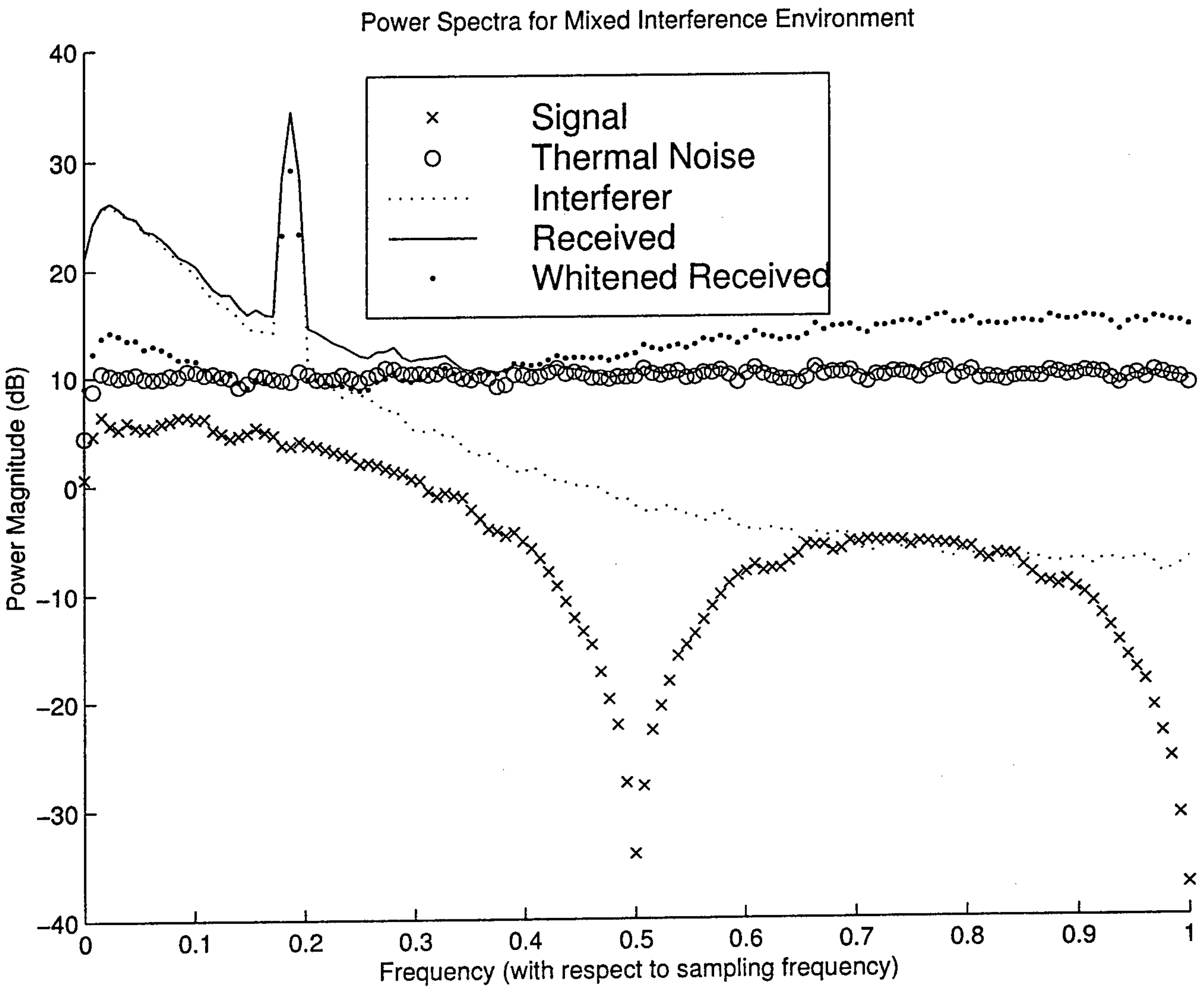


Figure 8: Mixed spectra for total ISR = 30dB with PG = 12dB

- Task 1: *Simulation of other interference environments*

  Although several scenarios have been investigated in this effort, other interference types exist that require study. Examples would be a bursty jammer, an FM modulated jammer and groups of narrowband jammers. The ARLO detector can be compared against the LO detection methods of previous work in these cases.

- Task 2: *Computational Efficiency*

  The methods of computation of the nonlinearity have been improved by the translation of MATLAB functions to C/C++. To implement LO detection within an embedded system, we must consider the issues of memory and processing speed. Eventually, the algorithms must be performed on a digital signal processor. As a next step to that process, the C algorithm development should be continued with the possibility of implementation on a DSP and real-time performance analysis and feasibility studies should be performed.

- Task 3: *Application of other LO techniques*

  The ARLO detector has been developed using simple histogram pdf estimation techniques. Previous work [1] suggests other means of estimating the nonlinearity could be more computationally efficient such as the FSA method. Enhancement techniques for high SNR environments can also be applied to the ARLO detector to increase robustness. For comparison, performance metrics can be derived for the ARLO detector as well.

- Task 4: *Projection onto convex sets (POCS)*

  Projection methods have been used to detect signals in narrowband interference environments with promising results. In particular, the advantage here could be the small number of samples needed for the

POCS algorithm in estimating the pdf, thus reducing the real-time processing. These methods should be investigated in conjunction with the ARLO detection scheme in other interference environments.

- Task 5: *Investigation of single CW interference*

  The ARLO detector does not perform as well as other detectors in the single CW environment. However, it outperforms them if even a second CW is added to the interference. This warrants further investigation into methods to improve its performance in the single CW case. Possible suggestions include varying the "P" parameter in coefficient estimation techniques or changing estimation methods altogether.

## References

[1] W.E. Jacklin, "Statistical Methods for Robust Locally Optimum Signal Detection," Ph. D. Dissertation , The Illinois Institute of Technology, Jul. 1996.

[2] D.R. Ucci, W.E. Jacklin, J.H. Grimm, *A Spread Spectrum Receiver With Nonlinear Processing*, Final Technical Report for Rome Laboratory, USAF Report No. RL-TR-93-50, 1993.

[3] D.R. Ucci and W.E. Jacklin *Robust Locally Optimum Detection in Auto-Regressive Noise*, First Status Report for Rome Laboratory, USAF RL Contract No. F30602-95-C-0215,

[4] D.R. Ucci and M.R. Mychal, *Robust Locally Optimum Detection in Auto-Regressive Noise*, Second Status Report for Rome Laboratory, USAF RL Contract No. F30602-95-C-0215, March 1997.

[5] D.R. Ucci and M.R. Mychal, *Robust Locally Optimum Detection in Auto-Regressive Noise*, Third Status Report for Rome Laboratory, USAF RL Contract No. F30602-95-C-0215, July 1997.

[6] D.R. Ucci and M.R. Mychal, *Robust Locally Optimum Detection in Auto-Regressive Noise*, Interim Status Report for Rome Laboratory, USAF RL Contract No. F30602-95-C-0215, November 1997.

[7] D.R. Ucci, M.R. Banys, A.L. Kaczmarski and M.R. Mychal and *Robust Locally Optimum Detection in Auto-Regressive Noise*, Fifth Status Report for Rome Laboratory, USAF RL Contract No. F30602-95-C-0215, February 1998.

[8] D.R. Ucci, A. Talathi and M.R. Mychal, *Robust Locally Optimum Detection in Auto-Regressive Noise*, Sixth Status Report for Rome Laboratory, USAF RL Contract No. F30602-95-C-0215, May 1998.

[9] D.R. Ucci, M.R. Mychal and A. Talathi *Robust Locally Optimum Detection in Auto-Regressive Noise*, Seventh Status Report for Rome Laboratory, USAF RL Contract No. F30602-95-C-0215, August 1998.

[10] P.M. Clarkson, *Optimal and Adaptive Signal Processing*, CRC Press, Inc., 1993.

[11] P.M. Clarkson and H. Stark, eds., *Signal Processing Methods for Audio, Images and Telecommunications*, London: Academic Press, 1995.

[12] M.H. Hayes, *Statistical Digital Signal Processing and Modeling*, New York: John Wiley and Sons, 1996.

[13] J.H. Higbie, "Adaptive Nonlinear Suppression of Interference,"*Proceedings of the 1988 IEEE MILCOM Conference*, pp. 23.3.1-9, 1988.

[14] S.A. Kassam, *Signal Detection in Non-Gaussian Noise*, New York: Springer-Verlag, 1988.

[15] J. Ketchum and J. Proakis, "Adaptive Algorithms for Estimating and Suppressing Narrowband Interference in PN Spread Spectrum Systems," *IEEE Transactions on Communications*, vol. COM-30 , pp. 913-923, May 1982.

[16] A.W. Maras, "Locally Optimum Bayes Detection in Ergodic Markov Noise," *IEEE Transactions on Information Theory*, vol. 40, pp. 41-45, Jan. 1994.

[17] J.L. Melsa and Cohn, D. L.,*Decision and Estimation Theory*, McGraw-Hill, Inc., 1978.

[18] D. Middleton, "Canonically Optimum Threshold Detection," *IEEE Transactions on Information Theory*, vol IT-12, pp. 230-243, Apr. 1966.

[19] L. Milstein and R. Iltis, "Signal Processing for Interference Rejection in Spread Spectrum Communications," *IEEE Acoustics, Speech and Signal Processing Magazine*, vol. 3, pp. 18-31, Apr. 1986.

[20] H. V. Poor, *An Introduction to Signal Detection and Estimation*, 2$^{nd}$ Ed., New York: Springer-Verlag, 1994.

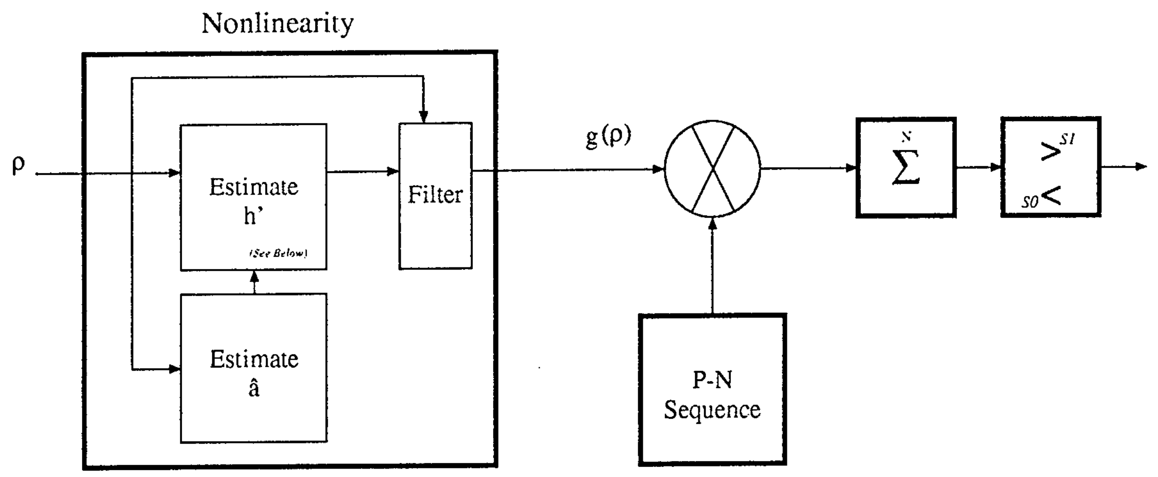
[21] W.H. Press et al., *Numerical Recipes in C: the Art of Scientific Computing*, Cambridge University Press, 1998.

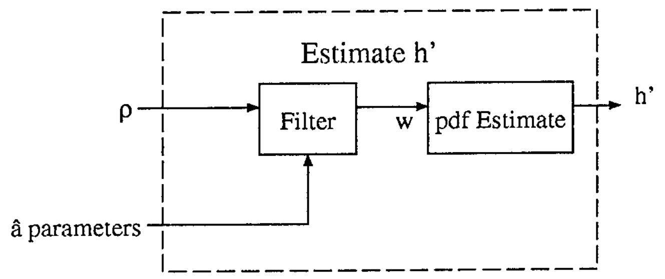
[22] J. G. Proakis, *Digital Communications*, Second Edition, New York: McGraw-Hill, 1989.

[23] D. Schilling et. al., "Spread Spectrum for Commercial Communications," *IEEE Communication Society Magazine*, pp. 66-79, Apr. 1991.

[24] H. Stark and J. Woods, *Probability, Random Processes, and Estimation Theory for Engineers*, Second Edition, Englewood Cliffs, NJ: Prentice-Hall, 1994.

## Appendix 1: ARLO Receiver Block Diagram





In the receiver block diagram shown above, $\rho$ is the received signal vector of length N, which is composed of both the signal sent by the transmitter and the noise present in the channel. The receiver first estimates the coefficients, $\{\hat{a}\}$, in the auto-regressive (AR) model using the modified covariance algorithm. Next, the AR model coefficients and the received signal are used to estimate $h'$, the *probability density function* (pdf) of the noise. This is currently accomplished via a histogram approximation method. The resulting LO nonlinearity, $g(\rho)$, is a function of N variables. This function is multiplied by a *pseudonoise* (P-N) sequence, applied to a summer, and then a decision is made as to whether a logical "0" or "1" was sent.

# Appendix 2: MATLAB© Simulation Code

```
% Program silmulation.m
% August 15, 1998
% Myron R. Mychal
%
% Simulates AR Model
% This m-file is a program to develop a robust locally optimum receiver for a
% communications system modeled in a colored noise channel.  This program
% requires the functions ar_histo.m and others.  The non-linear
% processor at the receiver is obtained from an autoregressive noise model
% using a histogram-based estimation technique for estimation of the noise pdf.
%
% William Jacklin, Myron Mychal
% Summer '96 - Summer '97

% Set parameters for simulation run - initialization procedure

sim_parms;          % Calls up the required parameters to run the simulation

% **************************
% **************************
% ** Begin simulation run **
% **************************
% **************************

tic;            % Start stopwatch to time one run

for snr = 1:1:length(Eb_sigma)    % Compute results for all Eb_sigma values
for pe_avg = 1:1:it      % Compute average of (it) iterations of Pe for each snr

% ******************************
% Implement the DSSS transmitter
% ******************************

% This simulation creates a BPSK signal of "B" bits sampled "Nb" times
% for each iteration of the BER tester

% "ref" is the BPSK modulated signal of randomly created information

ref = 2 * rand(1, B) - ones(1, B);      % Random BPSK data for 1 iteration: cts
ref = sign(ref);               % Data symbols at 1 sample per symbol
sym_amp = sqrt(sigmat2 * Eb_sigma(snr));  % Symbol amplitude assuming
                   %   chip-matched filtering

% IMPORTANT NOTE: Let the signal have amplitude of '1' and then DIVIDE
% the thermal noise by the symbol amplitude (below, in "Implement the Channel")

[ coded_data, chips ] = dsss(ref, Nc, B, PG, 1);    % Transmitted DSSS signal
                                      %   and PN chip sequence


% *********************
% Implement the channel
% *********************

% Choose the interferer type
if Itype == 'ar'
  [ interferer ] = th_inter(Ntot, ISR, a_t, iidtype, iidparms);    % Theoretical case
elseif Itype == 'pb'
  % [ interferer ] = pband(N, ISR, fc);       % Bandlimited interference
elseif Itype == 'ch'
  % [ interferer ] = chirp(N, ISR, fc);       % Chirped interference
elseif Itype == 'pl'
  % [ interferer ] = pulsed(N, ISR);       % Random pulsed interference
elseif Itype == 'fx'
  % [ interferer ] = fmxnoise(N, ISR);     % FM modulated interference
elseif Itype == 'mx'
  [ interferer ] = contwave(Ntot, ISRcw, I, freq) + th_inter(Ntot, ISRpb, a_t, iidtype,
iidparms);  % Mixed interferers;
elseif Itype == 'cw'
```

```
  [ interferer ] = contwave(Ntot, ISR, I, freq);   % CW interferers
end

% Add thermal noise with a Gaussian pdf of N(mu=0, sigmat^2) to the correlated
% interferer (and additive channel is assumed)

noise = (sigmat/sym_amp)*randn(1, Ntot) + interferer;

% ******************************
% Implement the observed signal
% ******************************

[ rcvd_tr ] = receiver(coded_data, noise, T, limit);

% *******************************************
% Implement a linear receiver with truncation
% *******************************************

% NO AR modeling (whitener) is necessary
% g(rho) = rho, so no histogram is necessary

lc_dec = decision(chips, rcvd_tr, Nb, B); % Make decisions
num_errs_lc = sum(0.5*abs(lc_dec - ref)); % Find no of errors for linear receiver
clear lc_dec;                  % Memory cleanup

disp(['Done with linear correlator.']);

% *********************************
% Implement LO detector techniques  % THE CRUX of this whole simulation
% *********************************

% #############################
% Find the appropriate whitener
% #############################

% We should loop this B/Bar times

a_hat = zeros(1, P+1);        % Initialize AR parameter estimate vector
[ a_hat error ] = modcov(rcvd_tr(1:Ntot/(B/Bar)), P); % Find the whitener

% ****************************************************
% Implement a linear correlator on the whitened signal
% ****************************************************

wc_dec = decision(chips, filter(-a_hat, 1, rcvd_tr), Nb, B);    % Make decisions
num_errs_wc = sum(0.5*abs(wc_dec - ref));        % Compute number of errors
clear wc_dec;                        % Memory cleanup
disp(['Done with linear correlator on whitened signal']);

g_lo = zeros(1, Ntot);     % Initialize the LO detector
g_ar = g_lo;          % Initialize the AR detector
g_lw = g_lo;          % Initialize the LO detector for the filtered case

Nsamh = Bh * Nb;      % Number of samples in each histogram

for histonum = 1:1:(B/Bh)  % Process Bh*Nb samples at a time for a histogram

% To keep track of which iteration the simulation is on, use the following display
%disp(['SNR = ', num2str(snr), ' It = ', num2str(pe_avg), ' Hist = ', num2str(histonum)]);

% ###############################
% Implement ARLO and LO receiver
% ###############################

block = zeros(1, Nsamh);       % Initialize block to prepare for the histogram
block = rcvd_tr(1, ((histonum - 1)*Nsamh+ 1):(histonum*Nsamh));% Apply the correct symbols
[ h_lo, bw_lo, min_lo ] = ar_histo(block, Nb, Bh, -1, K);    % LO (P = 0) a_0 = -1
[ h_wh, bw_wh, min_wh ] = ar_histo(block, Nb, Bh, a_hat, K);    % AR methods
```

```
clear block;              % Memory cleanup
whitened = filter(-a_hat, 1, rcvd_tr); % The whitened received signal
for symcnt = 1:Bh         % Apply nonlinearity to each symbol
  symbol = zeros(1, Nb);        % Look at one symbol
  symbolindx = ((histonum-1)*Nsamh+(symcnt-1)*Nb)+1 : ((histonum-1)*Nsamh+(symcnt)*Nb);
  symbolindx = symbolindx(:)';         % Force it to be a row vector
  symbol = rcvd_tr(symbolindx);        % Get the current symbol
  symbol = symbol(:)';          % Force it to be a row vector
  filsym = whitened(symbolindx); % The whitened symbol
  filsym = filsym(:)';          % Force it to be a row vector
  g_lo(1, symbolindx) = nlp(symbol, min_lo, Nb, h_lo, bw_lo, -1);
  g_lw(1, symbolindx) = nlp(filsym, min_wh, Nb, h_wh, bw_wh, -1);
  g_ar(1, symbolindx) = nlp(symbol, min_wh, Nb, h_wh, bw_wh, a_hat);
end

end                  % End of loop on "numhistos"

% ####################################
% Make ARLO and LO detector decisions
% ####################################

lo_dec = decision(chips, g_lo, Nb, B);     % Make decisions for LO
num_errs_lo = sum(0.5*abs(lo_dec - ref)); % Count no of errors for LO
clear g_lo, lo_dec;            % Memory cleanup

lw_dec = decision(chips, g_lw, Nb, B);     % Make decisions for LO on whitened
num_errs_lw = sum(0.5*abs(lw_dec - ref)); % Count no of errors for LO on whitened
clear g_lw, lw_dec;            % Memory cleanup

ar_dec = decision(chips, g_ar, Nb, B);     % Make decisions for ARLO
num_errs_ar = sum(0.5*abs(ar_dec - ref)); % Count no of errors for ARLO
clear g_ar, ar_dec;            % Memory cleanup

% Compute probability of bit error for this iteration and this "snr"

Pb_lc(pe_avg, snr) = num_errs_lc / B;      % Linear correlator
Pb_wc(pe_avg, snr) = num_errs_wc / B;      % Linear correlator on whitened
Pb_lo(pe_avg, snr) = num_errs_lo / B;      % LO memoryless detector
Pb_lw(pe_avg, snr) = num_errs_lw / B;      % LO detector on whitened signal
Pb_ar(pe_avg, snr) = num_errs_ar / B;      % ARLO detector with memory

end      % End of iteration on "pe_avg"
end      % End of iteration on "snr"

% Compute average Pb (from Monte Carlo analysis)

if (it == 1)               % If only one iteration
  Pb_lc_avg = Pb_lc;
  Pb_wc_avg = Pb_wc;
  Pb_lo_avg = Pb_lo;
  Pb_lw_avg = Pb_lw;
  Pb_ar_avg = Pb_ar;
else                  % If several iterations
  Pb_lc_avg = (1/it)*sum(Pb_lc);
  Pb_wc_avg = (1/it)*sum(Pb_wc);
  Pb_lo_avg = (1/it)*sum(Pb_lo);
  Pb_lw_avg = (1/it)*sum(Pb_lw);
  Pb_ar_avg = (1/it)*sum(Pb_ar);
end

run_time = toc;              % Stop stopwatch

save cwr08 Pb_lc_avg Pb_wc_avg Pb_lo_avg Pb_lw_avg Pb_ar_avg a_hat run_time;
%save cws08 noise rcvd_tr interferer;

% PLOT resulting Pb curves

%figure;
```

```
%hold on;
%semilogy(Pb_lc_avg, '-');
%title('Detector Performance in INSERT Interference Environment');
%xlabel('Signal-to-Thermal Noise Ratio (dB)');
%ylabel('Probability of Bit Error');
%semilogy(Pb_wc_avg, 'c-');
%semilogy(Pb_lo_avg, 'r-');
%semilogy(Pb_lw_avg, 'w-');
%semilogy(Pb_ar_avg, 'g-');
%legend(gca, 'Linear Correlator', 'Correlator on whitened', 'LO detector', 'LO on
whitened', 'ARLO'';
%hold off;

end
```

```
% This is a data file that contains the simulation parameters for the ARLO detector
% The file "simulation.m" calls this file and some of the functions within
% "simulation.m" require these global definitions.  See the individual functions
% for comments on where each parameter is used
% Book-keeping parameters

fname = 'ARresults';    % Output filename
it = 1;             % Number of Monte Carlo iterations to average for Pe calculation

% pdf estimation technique parameters
% - also see ar_histo.m, lo_histo.m & receiver.m

Bh = 512;           % Total number of symbols to calculate histogram
K = 33;             % Data points to calulate (# of breakpoints)
T = 1000;           % Range of support for pdf

% Spread-spectrum parameters

Nc = 4;             % Samples per chip
PG = 16;        % Processing gain (chips per bit)
Nb = PG*Nc;         % Samples per symbol (assume symbol is a bit)
B = Bh;             % Symbols transmitted/received and sent to BER tester
Ntot = B * Nb;      % Total number of samples in run

% AR model parameters

Bar = Bh/64;            % Total number of symbols to estimate AR parameters
Bg = Bh;            % Total number of symbols to calculate nlp
ar_method = 'mcov';     % Coefficient estimation method
                    % - 'mcov' = Modified Covariance
                    % - 'cov'  = Covariance
                    % - 'yw'   = Yule Walker
                    % - 'burg' = Burg Method

% Signal parameters and thermal noise
% - also see receiver.m

sigmat = 1;         % Standard deviation of thermal noise
sigmat2 = sigmat^2;     % Variance of thermal noise
Eb_min = -10;           % Minimum value of thermal SNR in db
Eb_max = 0;                         % Maximum value of thermal SNR in db
Eb_sigma = Eb_min:Eb_max;
for index = 1:length(Eb_sigma)
   Eb_sigma(index) = 10^(Eb_sigma(index)/10);    % Vector of SNR's
end
limit = 'hard';             % Limiter on noise
            % - 'hard' = noise blanker
            % - 'soft' = soft limiter
clear index;

% Correlated interferer parameters

ISR = 30;           % Interferer-to-signal ratio (db)
Itype = 'cw'                        % Interference type
            %   'cw' = Continuous Wave
            %   'ar' = Theoretical AR
            %   'pl' = Pulsed
            %   'pb' = Partial Band
            %   'ch' = Chirped FM
            %   'fx' = FM Modulated Noise
                                    %   'mx' = mixed partial band and CW

% Continuous wave jammers

if (Itype == 'cw') | (Itype == 'mx')
 I = 1;                             % Number of CW interferers
 freq = [ Nc*B ];                   % CW Interferer frequencies
 P = 16;                            % Number of AR coefficients
```

```
end

% Wide-band filtered white-noise jammers

if (Itype == 'ar') | (Itype == 'mx')
poly = [ 1 -0.75];        % The pole of the low-pass filter
L = 4;              % For a filter of order 2^L
for i = 1:L
 poly = conv(poly, poly);  % Recursively increase the order of the filter to 2^L
end
clear i;
a_t = -poly;
P = length(a_t) - 1;      % The number of coefficients
iidtype = 'gs';           % The type of iid noise sequence
            % - 'gs' = Gaussian
            % - 'pp' = Poisson
            % - 'lp' = Laplacian
            % - 'un' = Uniform
            % - 'mg' = Mixed Gaussian
            % - 'cy' = Cauchy
            % - 'rl' = Rayleigh
            % - 'ri' = Ricean
            % - 'nk' = Nakagami-m
iidparms = [ 0 1 ];       % Parameter vector for iidtype
            % - for 'gs' = [ mean standarddev ]
            % - for 'pp' = [ rate_parameter ]
            % - for 'lp' = [ sigma ]
            % - for 'un' = [ mean length ]
            % - for 'mg' = [ mean1 stddev1 mean2 stddev2 ]
            % - for 'cy' = [ mean, scale_parameter ]
            % - for 'rl' = [ ]
            % - for 'ri' = [ ]
            % - for 'nk' = [ ]
end

% Mixed jammers
if (Itype == 'mx')
   ISRcw = ISR/2;
   ISRpb = ISR/2;
end
```

```
% This M-file develops the decision statistic for the various types of
% receivers in a communication system.  It assumes that some function
% of the received signal, g(rho), is used in a direct sequence spread
% spectrum environment and multiplied by a chip sequence.  This is then
% passed through a discrete integrator (a summer) and sampled at the
% conclusion of the symbol (after Nb samples have passed).  The decision
% assumes synchronous BPSK modulation.
%
% Inputs:
%
%     pn - The PN chip sequence
%     g  - The detector function, g(rho)
%     Nb - The number of sample points per symbol
%     B  - The number of symbols
%
% Outputs:
%
%     dec   - The decision vector

function [ dec ] = decision(pn, g, Nb, B)

dec = ones(B ,1);        % Assume that all symbols are "1's"
stat = sum(reshape(pn .* g, Nb, B))';  % Sum each symbol correlated with the pn
H0 = find(stat <= 0);          % ML estimation choosing where "-1's" are
dec(H0) = -ones(size(H0));     % CHange symbols accordingly
dec = dec(:)';           % Make it a row vector
end;
```

# Appendix 3: C Simulation Code

## Appendix

```
/* filter.c
 * Version 1.0, 3/22/98
 *    Written by Michael Banys & Angela Kaczmarski
 *
 * Revision Control:
 *
 *
 *
 */

/* *******************************************************/
 * FILTER Function
 * OUTPUT = filter( b[M], a[N], data_in[] )
 *
 * b[M] = coefficients of numerator of impulse response
 * a[N] = coefficients of denominator of impulse response
 *      = order of filter
 *
 * This function computes the convolution
 * of vector "data_in" with the causal filter
 * whose time-domain filter coefficients
 * are of the following format:
 *
 *    a(1)*y(n) = -SUM(k=1,N) [ a(k)*y(n-k) ]
 *             +SUM(k=0,M) [ b(k)*x(n-k) ]
 *
 * If a(0) != 1, FILTER will normalize the filter
 * coefficients by a(0). If a(0)==0, return ERROR.
 *
 */


#define NULL    '0L'
#define ORDER_ERROR  "Filter numerator order is greater than denominator
                     order\n"

double filter( double b[], double a[], double data_in[], double y[])
{

   int   filter_order,        /* Order of acting filter */
         filter_iteration=0,  /* Variable tracking loop through input data */
         element_num,         /* Variable indicating current coefficient and
                              delay being processed */
      i=0,                    /* Iteration variable */
      M,                      /* Order of Numerator */
      N,                      /* Order of Filter (Denominator) */
      INPUT_LENGTH;           /* Length of input data vector */


   double   normalization_value,    /* a[0] coefficient value */
            *filter_delay,          /* Filter delay values */
```

```
        sumx,                       /* Temporary variable representing sum of
                                    elements of a[] multiplied by elements of
                                    filter_delay[]            */
        sumy;                       /* Temporary variable reperesenting sum
                                    of elements of b[] multiplied by elements
                                    of filter_delay[]         */


/* Determine number of elements in array b[] */
for (i=0; b[i] != NULL; i++)
   ;
M = i;

/* Determine number of elements in array a[] */
for (i=0; a[i] != NULL; i++)
   ;
N = i;

/* Determine number of elements in array data_in[] */
for (i=0; data_in[i] != NULL; i++)
   ;
INPUT_LENGTH = i;

filter_delay = malloc(N * sizeof(double) );  /* Allocate memory for
                                                filter_delay length */

filter_order = N;          /* Set order of filter */

/* Check to see if the length of b[] < length of a[]
   If length b[] >= length a[], output error message and exit function */
if (M >= N)
{
   printf(ORDER_ERROR);
   return 0;
}


/* Check to see if denominator of IR is normalized (i.e. does a[0]=1
   If denominator is not normalized, normalize IR by a[0] */
if (a[0]!= 1)
{  normalization_value=a[0];

   /* Normalize denominator */
   for (i=0; i<N; i++)
      a[i]=a[i]/normalization_value;

   /* Normalize numerator */
   for (i=0; i<M; i++)
      b[i]=b[i]/normalization_value;
}


/* Set w[n] delay array, filter_delay[], to all zeroes */
   for (i=0; i<=N; i++)
      filter_delay[i]=0;
```

```
    /* Start filtering process */

    for (filter_iteration=0; filter_iteration<INPUT_LENGTH;
       filter_iteration++)
    {
       sumx=sumy=0.0;     /* Initialze variables */

       /* Sum left side of Direct Form II Structure */
       for(element_num=1; element_num<N; element_num++)
          sumx = sumx - a[element_num] * filter_delay[element_num];

       /* Sum right side of Direct Form II Structure */
       for(element_num=1; element_num<M; element_num++)
          sumy = sumy + b[element_num] * filter_delay[element_num];

       /* Compute current filter delay (filter_delay[0]) */
       filter_delay[0] = data_in[filter_iteration] + sumx;

       /* Compute output element, y(n) (y[filter_iteration]) */
       y[filter_iteration] = b[0] * filter_delay[0] + sumy;

       /* Delay each filter_delay[] element by one */
       for ( element_num=N-1; element_num>0; element_num-- )
          filter_delay[element_num]=filter_delay[element_num-1];

    }

    free(filter_delay);
    return 0;

}
```

```
/* hist.c
 * Version 1.0, 8/24/98
 *    Written by Michael Banys & Angela Kaczmarski
 *
 * Revision Control:
 *
 *
 *
 */

/* hist( double signal[], int bin_info, double freq_count[], int bin_loc[] )
 *
 *    freq_count  = vector containing frequency distribution of each bin
 *    bin_loc     = vector containing center location of each bin
 *
 *    signal      = signal (vector) from which to compute histogram
 *    bin_info = number of bins necessary for histogram
 *
 * NOTE: All vectors must be NULL terminated for size determination
 *
 * The histogram function bins the elements of signal into the number of bins
 * specified by bin_info.  The function returns the number of elements in
 * each bin and the center point of each bin.
 */


#include <stdio.h>
#include <stdlib.h>

#define  NULL  '0L'


void hist( double signal[], int bin_info, int freq_count[], double bin_loc[])
{
   int   signal_length,          /* length of signal vector */
      i,                         /* counting variable */
      num_bins,                  /* Number of bins */
      j,
      accumulator,
      *temp_freq_count;          /* Intermediate bin center locations */

   double   min_y,               /* Minimum signal value */
      max_y,                     /* Maximum signal value */
      temp_double,
      *temp_bin_loc,
      binwidth;                  /* Width of each bin */


   for (i=0; signal[i] != NULL; i++);      /* Determine length of signal[] */
   signal_length = i;


   min_y = signal[0];                      /* Initialize min_y */
   for (i=1; i < signal_length; i++)       /* Find actual min_y */
   {
      temp_double = signal[i];
      if (temp_double < min_y)
```

```
         min_y = temp_double;
   }

   max_y = signal[0];                        /* Initialize max_y */
   for (i=1; i < signal_length; i++ )        /* Find actual max_y */
   {
      temp_double = signal[i];
      if (temp_double > max_y)
         max_y = temp_double;
   }

   binwidth = (max_y - min_y) / bin_info;      /* Determine binwidth */

   temp_bin_loc = malloc( (bin_info+1) * sizeof(double) );
                                               /* Allocate memory for
                                               temp_bin_loc */
   for (i=0; i <= bin_info; i++)               /* Determine bin centers */
      temp_bin_loc[i] = min_y + binwidth*i;
   temp_bin_loc[(int)bin_info] = max_y;

   for (i=0; i < bin_info; i++)
      bin_loc[i] = temp_bin_loc[i] + (binwidth/ 2.0);


   temp_freq_count = malloc( (bin_info+1) * sizeof(int));


   for( i=0; i < signal_length; i++)         /* Fill first row of
                                             temp_freq_count with zeros */
      temp_freq_count[i] = 0;



   for( i=1; i<bin_info+1; i++)          /* Determine bin member info */
   {
      accumulator = 0;                   /* Initialize accumulator */
      for( j=0; j < signal_length; j++)
         if( signal[j] <= temp_bin_loc[i] )
            accumulator++;

      temp_freq_count[i] = accumulator;
   }

   for (i=0; i<bin_info; i++)            /* Determine exact bin member info */
      freq_count[i]=temp_freq_count[i+1]-temp_freq_count[i];


   free(temp_freq_count);          /* Free dynamically allocated memory */
   free(temp_bin_loc);

}
```

# Appendix 4: C++ Simulation Code

```
 '  Created By Abhijit Talathi for ECE 449
    Simulation of DSSS Communications System

    This header file contains the class definition for an Interferer
  . The interferer is derived from the class Signal
  ' Three types of interferers are available.
  / 1. Gaussian
  / 2. Uniform
  / 3. Continuous Wave

#ifndef INTERFERER_H
#define INTERFERER_H

#include"signal.h"

const int MAX_FREQ = 64000;
const int MAX_LEVEL = 100;

template<class T> class Interferer : public Signal<T>
{
    protected:
        //  Interferance to signal ratio
        double isr; //  Interferance to signal ratio

        //static const int MAX_FREQ = 64000; //this is not accepted by complier
        //static const int MAX_LEVEL = 100;  //so is globally defined

    public:
        //  Constructor
        Interferer(const long length, const double level = 1);
};

//  Constructor for interferer
template<class T> Interferer<T>::Interferer(const long length, const double level):Signal<T>(length)
{
    try
    {
        if(level > MAX_LEVEL )
            throw "error";
    }
    catch(...)
    {
        cout<<endl<<"Interferer : Bad argument for interferer (ISR is too large!)";
        exit(EXIT_FAILURE);
    }
    isr = level;
}

// Continuous wave Interferer
template<class T> class Contwave : public Interferer<T>
{
    public:
        //  Constructor
        Contwave(const long legth, const double isr, const double freq);     //Constructor

        //  Function does nothing for CW interferer as attenuation is accounted for
        //  in the constructor
        void attenuate(const double factor){}

        //  Overloaded >> operator for user defined CW interferer
        friend istream &operator >> (istream &stream,Contwave<T> &proto);
};

//  Constructor for the class Interferer
//  This may be modified later to take a number of frequencies and phases
template<class T> Contwave<T>::Contwave(const long length, const double level, const double freq):In
terferer<T>(length,level)
{
    double phase = 0.0, time, amp;

    //  Phase will be made a vector of phases if required
    amp = sqrt(2 * pow(10, (isr / 10)));
    double delta_t = 1.0 / length;  //  Time interval

    for(int i = 0; i < length; i++)
    {
        time = delta_t * i;
        data[i] = amp * sin (2 * PI * freq * time + phase);//   Sinusoidal interferer
    }
    name("Cont Wave Interferer");
```

```
//  Overloaded >> operator for user defined CW interferer
template <class T> istream &operator >> (istream &stream, Contwave<T> &o)
{
    long length;
    double level,freq;

    cout<<"Creating a Continuous wave interferer" <endl;
    cout<<"(This may be modified later for multiple interferers)"<<endl;

    cout<<"Enter the Number of samples"<<endl;
    stream>>length;

    cout<<"Enter the Interference to Signal Ratio (ISR)"<<endl;
    stream>>level;

    cout<<"Enter the frequency of the wave"<<endl;
    stream>>freq;

    try
    {
        if(level > MAX_LEVEL || freq > MAX_FREQ )
            throw "error";
    }
    catch(...)
    {
        cout<<endl<<"ContWave Inter: Bad arguments for CW interferer";
        exit(EXIT_FAILURE);
    }

    Contwave<T> Temp(length,level,freq);
    //Fill in the elements of the continuous wave interferer.
    o=Temp;

    return stream;
}

//  Theoretical Gaussian Interferer
template<class T> class Gaussinter : public Interferer<T>
{
    public:
        Gaussinter(const long legth, const double isr, const double mu = 0.0, \
                const double sigma = 1.0);

        friend istream &operator >> (istream &stream, Gaussinter<T> &proto);
        //  Overloaded >> operator for user defined Gaussian interferer
};

template<class T> Gaussinter<T>::\
Gaussinter(const long length, const double level, const double mu , const double sigma)\
:Interferer<T>(length,level)
{
    gaussgen(length, mu, sigma );
    attenuate(isr);
    name("Gaussian Interferer");
}

template<class T>istream &operator>>(istream &stream, Gaussinter<T> &o)
{
    long length;
    double level, mu, sigma;

    cout<<"Creating a Gaussian Noise Sequence"<<endl;

    cout<<"Enter the Number samples"<<endl;
    stream>>length;

    cout<<"Enter the Interference to Signal Ratio (ISR)"<<endl;
    stream>>level;

    cout<<"Enter the Mean of the Iinterferer"<<endl;
    stream>>mu;

    cout<<"Enter the Standard Deviation of the Interferer"<<endl;
    stream>>sigma;

    //  Exceptions for arguments are handled in the function Gaussgen()
    Gaussinter<T> Temp(length,level,mu,sigma);
    //Fill in the elements of the Gaussian.
    o=Temp;
    return stream;
}

// Theoretical uniform interference
template<class T> class Unifointer : public Interferer<T>
{
    public:
        Unifointer(const long legth, const double isr, const double mu = 0.0);

        //  Overloaded >> operator for user defined Uniform Interferer
        friend istream &operator >> (istream &stream, Unifointer<T> &proto);

};

template<class T> Unifointer<T>::Unifointer(const long length, const double level, \
                                            const double mu ):Interferer<T>(length,level)
{
    unifogen(length, mu);
    attenuate(isr);
    name("Uniform Interferer");
}

//  Overloaded istream operator for an object of type Unifointer
template<class T> istream &operator>>(istream &stream, Unifointer<T> &o)
{
    long length;
    double mean,level;

    cout<<"Creating a Uniform Interferer"<<endl;

    cout<<"Enter the Number of samples"<<endl;
    stream>>length;

    cout<<"Enter the ISR"<<endl;
    stream>>level;

    cout<<"Enter the Mean the Interferer"<<endl;
    stream>>mean;

    // Exceptions for arguments are handled in the function Uniforgen()

    //  Create an object of type Uniform Interference

    //  Exceptions for arguments are handled in the function Gaussgen()
    Unifointer<T> Temp(length,level,mean);
    //Fill in the elements of the continuous wave interferer.
    o=Temp;
    return stream;
}


#endif
```

```
    Created By Abhijit Talathi for ECE 449
 //  Simulation of DSSS Communications System

 //  This header file contains the class definition for Noise
 //  The Noise is derived from the class Signal
 //  Types of Noise,
 //  1. Gaussian
 //  2. Uniform

#ifndef NOISE_H
#define NOISE_H

#include"signal.h"

//  Class for Noise
template<class T> class Noise : public Signal<T>
{
    protected:
        long seed;  // Seed for Noise generation

    public:
        Noise(const long length);

};

template<class T> Noise<T>::Noise(const long length):Signal<T>(length)
{
    seed = (int)time(NULL);
    srand(seed);
}

//Gaussian Noise
template<class T> class Gaussnoise : public Noise<T>
{
    public:
        Gaussnoise(const long legth, const double mu = 0.0, const double sigma = 1.0);

        friend istream &operator >> (istream &stream, Gaussnoise<T> &proto);
        //Overloaded >> operator for user defined Gaussian Noise

};

template<class T> Gaussnoise<T>::Gaussnoise(const long length, const double mu, const double sigma):
 Noise<T>(length)
{
    gaussgen(length, mu, sigma );
    name("Gaussian Noise");
}

template <class T>istream &operator>>(istream &stream, Gaussnoise<T> &o)
{
    long length;
    double level, mu, sigma;

    cout<<"Creating a Gaussian Noise Sequence"<<endl;

    cout<<"Enter the Number samples"<<endl;
    stream>>length;

    cout<<"Enter the Mean the Noise"<<endl;
    stream>>mu;

    cout<<"Enter the Standard Deviation of the Noise"<<endl;
    stream>>sigma;

    //  Exceptions are handled in the gaussgen function

    Gaussnoise<T> Temp(length,mu,sigma);
    //Fill in the elements of the continuous wave interferer.
    o=Temp;
    return stream;
}

//Uniform noise
template<class T> class Unifonoise : public Noise<T>
{
    public:
        Unifonoise(const long legth, const double mu = 0.0);

        friend istream &operator >> (istream &stream, Unifonoise<T> &proto);
        //Overloaded >> operator for user defined Uniform Noise
};

template<class T> Unifonoise<T>::Unifonoise(const long length, const double mu):Noise<T>(length)
{
    unifogen(length, mu);
    name("Uniform Noise");
}

//Overloaded istream operator for an object of type Unifonoise
template<class T>istream &operator>>(istream &stream, Unifonoise<T> &o)
{
    long length;
    double mean;

    cout<<"Creating a Uniform Noise Sequence"<<endl;

    cout<<"Enter the Number of samples"<<endl;
    stream>>length;

    cout<<"Enter the Mean the Noise"<<endl;
    stream>>mean;

    Unifonoise<T> Temp(length,freq);
    //Fill in the elements of the continuous wave interferer.
    o=Temp;
    //  Exceptions are handled in the Gaussgen function
    return stream;
}

#endif
```

```
//  Created By Abhijit Talathi for ECE 449
//  Simulation of DSSS Communications System

//  This header file contains the class definition for a BPSK signal
//  The BPSK is derived from the class Signal
//  This cosists of a binary signal composed of the symbols -1 and 1

//  Class binary phase shift keyed signal (BPSK)(the message encoded in ones and -ones )
#ifndef BPSK_H
#define BPSK_H

#include"signal.h"

template<class T> class BPSK : public Signal<T>
{
    public:
        //  Constructor
        BPSK(const long length);

        //  Generates a BPSK signal
        void BPSKgen(const long length);

        //  Performs Direct Sequence Spread Spectrum codification of the BPSK signal
        Signal<T> dsss( Signal<T> &signal, const int NC,
                                        const int B, const int PG , const int amp);

        //  Overloaded >> operator for user defined BPSK signal
        friend istream &operator >> (istream &stream, BPSK &proto);
};

template<class T> BPSK<T> :: BPSK(const long length):Signal<T>(length)
{
    BPSKgen(length);
    name("BPSK");
}

template<class T> void BPSK<T> :: BPSKgen(const long length)
{
    for(int i=0; i < length; i++)
    {
        data[i] = (rand()%2) ? 1 : -1;// Initailize to ones and -ones
    }
}

//  Implement the function DSSS ie Discrete Spread Spectrum Signal on The BPSK signal

template<class T>
Signal<T> BPSK<T>::dsss( Signal<T> &signal,const int NC,
                                        const int B, const int PG , const int amp)
{
    ConvertToRowMatrix();    // Method of class Matrix

    int NB = NC * PG;    //  Constants for DSSS
    int N = NB * B;

    Matrix<T> tmp(NB,1,1),mat1;

    tmp=tmp *(*this);    //  Multiply the Matrices

    mat1=tmp;

    mat1.ConvertToRowMatrix();

    tmp = RandomMatrixOne(1, (B*PG))*2;

    tmp-=1; //  Subsract 1 from temp to get BPSK

    tmp.Sign(); //  Final BPSK signal

    tmp= Matrix<T>(NC,1,1)*tmp; //  Multiply by ones

    Matrix<T> chips=tmp;

    chips.Reshape(N,1); //  Invoke reshape in the parent

    chips.Sign();

    chips.ConvertToRowMatrix();

    Matrix<T> Temp = chips;    Create signal
    Temp.Indmult(matl);

    signal=Temp * amp;

    return chips;
}

//  Overloaded istream operator for an object of type BPSK
template<class T>istream &operator>>(istream &stream, BPSK<T> &o)
{
    long length;

    cout<<"Creating a Binary Phase Shift Keyed Sequence"<<endl;

    cout<<"Enter the Number of samples"<<endl;
    stream>>length;

    //  Create an object of type BPSK
    o.BPSKgen(length);

    return stream;
}

#endif
```

```
.  constructs the final received signal
template<class T>
Signal<T> Receiver(const Signal<T> receive, const Signal<T> chipseq, const long NB, const long B,con
st T limit)
{
    Signal<T> stat(B);

    stat=receive;

    //implement receiver for hard limiting
    for(int i=0;i<stat.NoofColumns();i++)
    {
         if((fabs(stat(0,i))) > limit) stat(0,i) = 0.0;
    }
    //  End of hard limiting

    //  Implement decision
    stat.Indmult(chipseq);

    stat.Reshape(NB,B);


    //cout<<stat;
    stat=stat.Sum();


    //Make decisions based on values of the elements of stat
    for(i=0;i<stat.NoofColumns();i++)
    {
        if (stat(0,i) <= 0.0) stat(0,i) = -1.0;
        else stat(0,i) = 1.0;
    }

    return stat;
}
```

```
//  Created By Abhijit Talathi for ECE 449
//  Simulation of DSSS Communications System

//  Header file for the definition of the class Signal
//  used in simulation of a Direct Sequence Spread Spectrum
//  communication system.
//  This header file defines the class signal which is derived from the
//  Matrix template class.

//Include the Matrix Teplate Definition file

#ifndef SIGNAL_H
#define SIGNAL_H

#include"Matrix.h"

//Macro for uniform random generator
#define unirand() (float) ((double)rand()/RAND_MAX)

const double PI = 3.1415927;


const int NAMELEN = 20; //  Longest possible name

template <class T> class Signal : public Matrix<T>
{
    private:

        //  A flag that is set once the maximum and minimum are found
        //  This allows us to find the bound only once if they are
        //  required
        short fndbound;

        static const int MU_MAX;     //  Maximum value of MU
        static const int SIG_MAX;    //  Maximum Value of Sigma
//      static const int NAMELEN;    //  Longest possible name

    protected:

        //  The maximum value of the signal
        T   max;

        //  The minimum value of the signal
        T   min;

        //  Stores the name of the signal
        char tag[NAMELEN];

        //  Sets the name (Only available to derived classes)
        void name(char* identity) {strncpy(tag,identity,NAMELEN);}

    public:

        //  Default constructor
        Signal(const long Length,const T mini = 0,const T maxi = 0);

        // Converting Matrix to signal
        Signal(const Matrix<T> mat);

        void findbounds(); // Find the boundaries

        T sigmax(); //  Set the maximum if necessary and provide to user

        T sigmin(); //  Set the minimum if necessary and provide to user

        //  Returns the name of the signal
        virtual char* identify() {return tag;}

        //  Function to change the amplitude by a fixed factor
        virtual void attenuate(const double factor);

        //  The basic types of Noise models
        //  Gaussian
        void gaussgen(const long length, const double mu = 0.0, const double sigma = 1.0);

        //  Uniform
        void unifogen(const long length, const double mu = 0.0);

        //  Swaps first half and second half of signal (similar to matlab)
        void fftshift();
    }
    catch(...)
    {
        cout<<endl<<"Gaussian Generator : Arguments are to large for this simulation";
        exit(EXIT_FAILURE);
    }

    double temp=0.0;
    double s=sigma / 3.33;//Adjusted Sigma

    for(int i = 0, n = 12; i < length; i++, n=12)
    {
        for(; n > 0; n--)
        {
            temp += unirand();//Call uniform random generator
        }
        temp =(s * (temp - 6));
        data[i]=temp;
    }
}

//  Generate a Uniformly Distributed signal
template<class T>
void Signal<T>::unifogen(const long length, const double mu)
{
    try
    {
        if(mu > MU_MAX)
            throw "error";
    }
    catch(...)
    {
        cout<<endl<<"Uniform Generator : Arguments are to large for this simulation";
        exit(EXIT_FAILURE);
    }
    //  Generate a uniform random distribution with mu as the mean
    for(int i = 0; i < length; i++)
    {
        data[i] = (T)(rand() - 0.5 + mu);//Make it uniform distribution with mean = mu
    }
}

//  Swaps the first half of the signal with the second half
//  Similar to the function in matlab which does the same.
template<class T>void Signal<T>::fftshift()
{
    int midpt;
    if(col%2)
    {
        midpt = (int)(col / 2) + 1;//Center as defined by the required function
    }
    else
    {
        midpt = (col / 2);
    }
    //Divide the signal into two parts and swap each part
    for(int i = 0; i < midpt; i++)
    {
        temp=data[i];
        data[i]=data[i+midpt];
        data[i+midpt]=temp;
    }
}


template<class T>const int Signal<T>::MU_MAX = 10;   //  Maximum value of MU
template<class T>const int Signal<T>::SIG_MAX = 5;   //  Maximum Value of Sigma
//const int Signal::NAMELEN = 20;        //  Longest possible name

#endif
```

```
// Created by Dhiraj Gandhi
// Simulation of DSSS Communication System
// Matrix class
//===================

#ifndef MATRIX_H   // To avoid multiple inclusion of the same class
#define MATRIX_H

#include<iostream.h>
#include<stdlib.h>
#include<math.h>
#include<assert.h>
#include<string.h>
#include<time.h>

#define EQUAL 0
#define NOTEQUAL 1



// Matrix Class Definition
template<class T>
class Matrix
{
    protected:
        // Rows --> No of rows in the matrix
        // Clumns --> No of columns in the matrix
        // Data  ---> Will point to starting address of array
        // Rows and columns sometimes may greater than limit of int  hence chosen as long

        long rows,columns;
        T* data;

    public:
        // Default constructor with long row , columns and value equall to zero
        Matrix(const long row=0,const long col=0,const T value=0);

        // Copy constructor
        Matrix(const Matrix<T>& mat);

        // Constructor taking the char* as a arguemnet and converts into the
        // required type
        // This constructor can convert  ("2 2 2,2 2 2,2 2 2 ") into 3x3 matrix of 2
        Matrix(const char * str);

        // Destructor
        ~Matrix()
        {
            delete []data;
        }

        // Return Total No of Rows of the matrix
        long NoofRows()
        {   return rows; }

        // Returns Total no of columns of the matrix
        long NoofColumns()
        {   return columns; }

        // Overloading () operator for indexing
        T operator()(const long row,const long col)const
        {
            try
            {
                if(row>=rows && row <0 && col >=columns && col<0)
                    throw "error";

            }
            catch(...)
            {
                cout<<endl<<" Row no.and Column no should be less than \
                            total rows and columns"<<endl;
                return 0;
            }
            return data[(row*columns)+col];
        }
```

```
 / Overloading () operator for indexing (reference)
T& operator()(const long row,const long col)
{
    try
    {
        if(row>=rows && row <0 && col >=columns && col<0)
            throw "error";
    }
    catch(...)
    {
        cout<<endl<<" Row no.and Column no should be less than \
                total rows and columns"<<endl;
        return data[0];

    }
    return data[(row*columns)+col];
}


// Overloading composite operators
Matrix<T>&  operator+= (const Matrix<T> mat);
Matrix<T>&  operator-= (const Matrix<T> mat);

// Overloading composite operators for value
Matrix<T>&  operator+= (const T value);
Matrix<T>&  operator-= (const T value);
Matrix<T>&  operator*= (const T value);
Matrix<T>&  operator/= (const T value);

// Overlaoding binary operators
Matrix<T> operator-()const;
Matrix<T> operator+ (Matrix<T> mat);
Matrix<T> operator- (Matrix<T> mat);
Matrix<T> operator* (const Matrix<T> mat)const;
Matrix<T> operator+ (const T value)const;
Matrix<T> operator- (const T value)const;
Matrix<T> operator* (const T value)const;
Matrix<T> operator/ (const T value)const;


// Overloadind = operator
Matrix<T>& operator= (const Matrix<T> mat);
Matrix<T>& operator= (const T value);


// Overloading comparisons operator for matrix
short operator == (const Matrix<T> mat)const;
short operator != (const Matrix<T> mat)const;

// Overloading comparison operators for value
short operator == (const T value)const;
short operator != (const T value)const;

// Multiplies individual elements of the matrix
void  Indmult(const Matrix<T> mat);


// Calculate inverse of given matrix
Matrix<T> Inverse();

// Calculate the determinant value of given square matrix
T Det();

// Returns the submatrix part of given matrix specified by
//  startrow,startcol and endrow,endcol
Matrix<T> SubMatrix(const long startrow,const long startcol,\
                        const long endrow, const long endcol);

// Reshapes the matrix with new rows and columns
 void Reshape(const long newrow, const long newcol);

//Returns the signs of the  each element of the Matrix
 void Sign();

// Converts the Matrix into the row Matrix
 void ConvertToRowMatrix();
```

```
        // Converts the Matrix into the Column Matrix
        void ConvertToColumnMatrix();

        // Swaps the Column first with Column second
        void SwapColumns(const long first,const long second);

        // Swaps the Row first with Row second
        void SwapRows(const long first,const long second);

        // Discrete Fourier Transform
        Matrix<T> Dft();

        // Convolution function
        Matrix<T> Convo(const Matrix<T> mat);

        // Sum of the all elements of given matrix
        Matrix<T> Sum();

        // Finding the no of errors between two given matrices
        long  NoofErrors(const Matrix<T> mat);

        // Polymorphic function for addition
        virtual Matrix<T> add(const Matrix<T> mat);

        // Overloading << and >> operators
        friend ostream& operator<<(ostream& os,Matrix<T>& cstr); // Overloading << operator
        friend istream& operator>>(istream& is,Matrix<T>& cstr); // Overloading >> operator

};

// Default constructor with long row , columns and value equall to zero
template<class T>Matrix<T>::Matrix(const long row,const long col,const T value)
{
    rows=row;
    columns=col;
    try
    {
        if((data=new T[rows*columns])==NULL)
            throw "error";
    }
    catch(...)
    {
        cout<<endl<<" Memory Allocation Error"<<endl;

    }
    // temp is temporary variable that we can use to avoid same multiplications in the loop
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
                data[i]=value;
}


// Copy constructor
template<class T>Matrix<T>::Matrix(const Matrix<T> &mat)
{
    rows=mat.rows;
    columns=mat.columns;
    long temp=rows*columns;
    try
    {
        if((data=new T[temp])==NULL)
            throw "error";

    }
    catch(...)
    {
        cout<<endl<<" Memory Allocation Error"<<endl;

    }
    for(long i=0;i<temp;i++)
            data[i]=mat.data[i];

}

// Constructor taking char * as arguement
// This constructor can convert  ("2 2 2,2 2 2,2 2 2 ") into 3x3 matrix of 2
template<class T>Matrix<T>::Matrix(const char *str)
{
    int long MAXDIGIT=4;  // No of max digits of the number
    long length=strlen(str);
    long rowcnt=0,columncnt=0;
    columns=0;


    // This loop will determine total number of columns and rows
    for(long i=0;i<length;i++)
    {
        if(str[i]==' ')
            columncnt++;
        if(str[i]==',')
        {
            rowcnt++;
            columncnt++;
            if(columns ==0){
                columns=columncnt;
            }
            try
            {
                if(columns != columncnt)
                        throw "error";
            }
            catch(...)
            {
                cout<<endl<<" No of columns are not equal for every row"<<endl;

            }
            columncnt=0;
        }
    }

    rows=rowcnt+1;
    data= new T [rows*columns];
    long datacnt,negative=1,convertcnt=0;
    T value=0,convert[MAXDIGIT];
    for(int j=0;j<MAXDIGIT;j++)
                convert[j]=0;


    // This will take each individual member and convert it into the required type
    for(i=0,datacnt=0;i<length;i++)
    {

        if(str[i]==' ' || str[i] == ',')
        {

            value=0;
            for(int m=MAXDIGIT-1;convert[m]==0 && m >=0;m--);
            for(j=0;j<=m;j++)
            {
                value+=(convert[j]*pow(10,(m-j)));
                convert[j]=0;
            }
            cout<<endl;
            data[datacnt]=negative*value;
            datacnt++;
            convertcnt=0;
            negative=1;
            continue;
        }
        if(str[i]=='-'&&(str[i-1]==' ' || str[i-1] == ','))
        {
            negative=-1;
            continue;
        }
        convert[convertcnt]=str[i]-'0';
        convertcnt++;

    }

}

// Overloading += operator for matrix
template<class T>Matrix<T>& Matrix<T>::operator+= (const Matrix<T> mat)
{
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
            data[i]+=mat.data[i];;
    return *this;

}
```

```
 / Overloading -= operator for matrix
template<class T>Matrix<T>& Matrix<T>:: operator-= (const Matrix<T> mat)
{
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             data[i]-=mat.data[i];;
    return *this;
}

// Overloading += operator for value
template<class T>Matrix<T>& Matrix<T>::operator+= (const T value)
{
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             data[i]+=value;
    return *this;

}

// Overloading -= operator for value
template<class T>Matrix<T>& Matrix<T>::operator-= (const T value)
{
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             data[i]-=value;
    return *this;

}

// Overloading *= operator for value
template<class T>Matrix<T>& Matrix<T>::operator*= (const T value)
{
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             data[i]*=value;
    return *this;

}

// Overloading /= operator for value
template<class T>Matrix<T>& Matrix<T>::operator/= (const T value)
{
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             data[i]/=value;
    return *this;

}

// Oveloading Unary - operator
template<class T>Matrix<T> Matrix<T>::operator-()const
{
    Matrix<T> result(rows,columns);
    for(long i=0;i<(rows*columns);i++)
             result.data[i]=-data[i];
    return result;
}


// Overloading * operator for Matrix
template<class T>Matrix<T> Matrix<T>::operator* (const Matrix<T> mat)const
{
    try
    {
        if( columns != mat.rows)
            throw "error";

    }
    catch(...)
    {
        cout<<endl<<" Multiplication: Columns of first Matrix = Rows of second matrix"<<endl;
        return Matrix<T>();
    }
    long temp;
    Matrix<T> result(rows,mat.columns);
    for (long i=0; i<rows; i++)
    {
        temp=i*columns;
        for (long j=0; j<mat.columns; j++)
        {
            for (long k=0; k<columns; k++)
            {
                result(i,j) += data[temp+k] * mat(k,j);
            }
        }
    }
    return result;
}


// Overlading + operator for value
template<class T>Matrix<T> Matrix<T>::operator+(const T value)const
{
    Matrix<T> result(rows,columns);
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             result.data[i]=data[i]+value;
    return result;
}


// Overloading - operator for value
template<class T>Matrix<T> Matrix<T>::operator-(const T value)const
{
    Matrix<T> result(rows,columns);
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             result.data[i]=data[i]-value;
    return result;
}

// Overloading * operator for value
template<class T>Matrix<T> Matrix<T>::operator*(const T value)const
{
    Matrix<T> result(rows,columns);
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             result.data[i]=data[i]*value;
    return result;
}

// Overloading / operator for value
template<class T>Matrix<T> Matrix<T>::operator/(const T value)const
{
    Matrix<T> result(rows,columns);
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             result.data[i]=data[i]/value;
    return result;
}

// Overloading = operator for matrix
template<class T>Matrix<T>& Matrix<T>::operator= (const Matrix<T> mat)
{
    // If the size if the matrix is not matching with the new matrix then
    // delete the old matrix and create the new matrix
    if(rows != mat.rows || columns!= mat.columns)
    {
        rows=mat.rows;
        columns=mat.columns;
        delete []data;
        try
        {
            if((data= new T[rows*columns])==NULL)
                throw "error";
        }
        catch(...)
        {
            cout<<endl<< " Memory Allocation error"<<endl;

        }
    }
    long temp=rows*columns;
    for(long i=0;i<temp;i++)
             data[i]=mat.data[i];
    return *this;
```

```
                    throw "error";
      }
      catch(...)
      {
          cout<<endl<<" nsolve : Fails on the given Matrix";
          delete []d;
          return Matrix<T>();
      }
      nrow=i-1;
      for (j=0;j<rows;j++)
      {
              nrow=(nrow+1)%rows;
              result(nrow,i)=*(d+j*(columns+1)+columns);
      }
   }
   delete []d;
   return result;
 }


// This function is internally called by Matrix inverse
// downloaded from the web
template <class T> long nsolve(long rows,T *data)
{
  long i,j,k;
  long cols;
  T dtemp;
  cols=rows+1;
  for (i=0;i<rows;i++)
     {
     for (j=i;j<rows&&*(data+j*cols+j)==0.0;j++);
     if (*(data+j*cols+j)==0.0)
     {
       cout<<endl<<"nsolve error: singular matrix";
       return 1;
     }
     if (j!=i)
     {
       for (k=0;k<cols;k++)
       {
         dtemp=*(data+i*cols+k);
         *(data+i*cols+k)=*(data+j*cols+k);
         *(data+j*cols+k)=dtemp;
        }
     }
     for (j=cols-1;j>=0;j--)
     {
             *(data+i*cols+j) /= *(data+i*cols+i);
     }
     for (j=i+1;j<rows;j++)
     {
             for (k=cols-1;k>=i;k--)
                 *(data+j*cols+k)-=*(data+j*cols+i) * *(data+i*cols+k);
     }
         }
     for (i=rows-2;i>=0;i--)
     {
         for (j=cols-2;j>i;j--)
         {
             *(data+i*cols+cols-1)-= \
             *(data+i*cols+j) * *(data+j*cols+cols-1);
             *(data+i*cols+j)=0;
         }
     }
  return 0;
}


// finding the value of determinant
template<class T> T Matrix<T>:: Det()
{
  T p1,p2,p3,d;
  long i,j,k;
  try
  {
        if (columns!=rows)
            throw "error";
  }
  catch(...)
  {
        cout<<endl<<" Det: The Matrix should be square matrix"<<endl;
        return 0;
    }
    d=0;
    for (i=0;i<columns;i++)
    {
          p1=p2=1;
          p3=data[i];
          k=i;
          for (j=1;j<columns;j++)
          {
              k=(k+1)%columns;
              p1*=data[(j*columns)+k];
              p2*=data[(columns-j)*columns+k];
          }
          p3*=(p1-p2);
          d+=p3;
      }
   return d;
}


// Returning the submatrix within the matrix specified by the start and end rows columns
template<class T> Matrix<T> Matrix<T>::SubMatrix(const long startrow, const long startcol
                                                ,const long endrow,const long endcol)
{

    try
    {
        if( endrow < startrow || endcol < startcol)
            throw "error";
    }
    catch(...)
    {
        cout<<endl<<" Submatrix: Ending row and ending column should be greater \
                            than stating row and starting col"<<endl;
        return Matrix<T>();
    }
    Matrix<T> result((endrow-startrow+1),endcol-startcol+1);
    for(long i=startrow,p=0;i<=endrow;i++,p++)
        for(long j=startcol,q=0;j<=endcol;j++,q++)
            result(p,q)=data[i*rows+j];
    return result;
}


// Reshapes the matrix with new rows and new columns
template<class T>void Matrix<T>::Reshape(const long newrow, const long newcol)
{
    try
    {
        if((newrow*newcol)!=(rows*columns))
            throw "error";
    }
    catch(...)
    {
        cout<<endl<< " Reshape: Total no of elements in old and new matrix should match"<<endl;

    }
    T* dummy = new T[rows*columns];
    long temp=newrow*newcol;

    // This Reshape routine will store the elements columnwise as desired by Matlab
    for(long i=0,k=0;i<columns;i++)
        for(long j=0;j<temp;j+=columns,k++)
            dummy[k]=data[i+j];
    for(i=0,k=0;i<newrow;i++)
        for(long j=0;j<temp;j+=newrow,k++)
                data[k]=dummy[i+j];

    rows=newrow;
    columns=newcol;
    delete []dummy;
}


// For each element of the matrix Sign gives
// 1 if the element  is greater than zero,
// 0 if it equals zero
//-1 if it is  less than zero
```

```
    cout<<" wn="<<wn<<endl;
    float  arg,c,s;
    for(long k=0; k<len;k++)
    {
        kml=k;
        for(long i=0;i<len;i++)
        {
            iml=i;
            arg=wn*kml*iml;
            c=cos(arg);
            s=sin(arg);
            result.data[k]+=data[i]*c;
        }
    }
    return result;
}


//Convolving the given matrix with provided matrix
template<class T> Matrix<T> Matrix<T>::Convo(const Matrix<T> mat)
{
    long len1=rows*columns;
    long len2=mat.rows*mat.columns,k;
    Matrix<T> result(1,(len1+len2-1));
    for(long i=0;i<len1;i++)
    {
        for(long j=0;j<len2;j++)
        {

            k=i+j;
            result.data[k]+=data[i]*mat.data[j];
        }
    }
    return result;
}


// Sum all the elements of given matrix Columwise
template<class T> Matrix<T> Matrix<T>::Sum()
{
    Matrix<T> result(1,columns,0);
    long j;
    for(long i=0;i<columns;i++)
        for(j=0;j<rows;j++)
            result.data[i]+=data[j*columns+i];
    return result;
}

// Finding the no of errors between two given matrices
template <class T> long Matrix<T>::NoofErrors(const Matrix<T> mat)
{

    long errors=0;
    try
    {
        if((mat.rows*mat.columns)!=(rows*columns))
            throw "error";
    }
    catch(...)
    {
        cout<<endl<< " NoofErrors: Total no of elements in old and new matrix should match"<<endl;
        return errors;

    }
    long temp=rows*columns;

    for(long i=0;i<temp;i++)
        if(data[i] != mat.data[i])
                errors++;
    return errors;

}



// Overloading << operator
template <class T> ostream& operator<<(ostream& os,Matrix<T>& mat)
{
    for(long i=0;i<mat.rows;i++)
    {
     Created By Dhiraj Gandhi & Abhijit Talathi for ECE 449
     Simulation of DSSS Communications System
     Main simulation program

#include "bpsk.h"
#include "interferer.h"
#include "noise.h"

 // constructs the final received signal
template<class T>
Signal<T> Receiver(const Signal<T> receive, const Signal<T> chipseq, const long NB, const long B,con
st T limit)
{
     Signal<T> stat(B);

     stat=receive;

     //implement receiver for hard limiting
     for(int i=0;i<stat.NoofColumns();i++)
     {
          if((fabs(stat(0,i))) > limit) stat(0,i) = 0.0;
     }
     //  End of hard limiting

     //  Implement decision
     stat.Indmult(chipseq);

     stat.Reshape(NB,B);


     //cout<<stat;
     stat=stat.Sum();


     //Make decisions based on values of the elements of stat
     for(i=0;i<stat.NoofColumns();i++)
     {
         if (stat(0,i) <= 0.0) stat(0,i) = -1.0;
         else stat(0,i) = 1.0;
     }

     return stat;
}

int main()
{
    const int B = 1028; // Symbols transmitted and received
    const int NC = 4;     // Samples per chip
    const int PG = 16;  // processing gain
    int ISR;    //Interference to signal Ratio

    //  To store the coded signal
    Signal<double> Codedsig(B*NC*PG);

    //  Type signal to store the received signal
    Signal<double> Received(B*NC*PG);

    cout<<"This is a Demo for specified SNR and ISR "<<endl;
    cout<<"Errors are calculated for specified values of ISR "<<endl;



    int SNR;

    do
    {
        cout<<endl<<" Enter the Signal to noise ratio( 0 - 100 dB) : ";
        cin>>SNR;
    }while(SNR <0 || SNR > 100);

    do
    {
        cout<<endl<<" Enter the Interference to signal ratio (0- 100) : ";
        cin>>ISR;
    }while(ISR <0 || ISR > 100);


    double Noisefactor = pow(10,(SNR/10));
    /// Create a BPSK signal
    BPSK<double> Transsig(B);

    //  To store the chip sequence
    Signal<double> chips(PG);

    //  Codedsig is the signal to be transmitted.
    //  It is created from  Transsig by applying Discrete Spread Spectrum techniques
    //  Chips is the chips sequence.

    chips=Transsig.dsss(Codedsig,NC,B,PG,1);

    //  Generate a Continuous Wave Interferer
    Interferer<double> *Int;

    //  Generate a Gaussian Noise Sequence.
    Gaussnoise<double> GN(B*NC*PG);

    //Multiply the Noise by the Reciprocal of the SNR
    GN *= (1/Noisefactor);

    cout<<"Choose the type of interferer"<<endl;
    cout<<"1 : Continuous Wave Interferer"<<endl;
    cout<<"2 : Gaussian Interferer"<<endl;
    cout<<"3 : Uniform Interferer"<<endl;

    short choice = 0;
    while(choice >3 || choice <1)
            cin>>choice;
    Contwave<double> CW(B*NC*PG,ISR,NC*B);
    Gaussinter<double> GI(B*NC*PG,ISR,0,1);
    Unifointer<double> UI(B*NC*PG,ISR,0);

    // Get the interfererence and add the noise
    switch (choice)
    {
        case 1:
            {
                Int=  &CW;
                break;
            }
        case 2:
            {
                Int = &GI;
                break;
            }
        case 3:
            {
                Int = &UI;
                break;
            }
    }

    // Add will polymorphically add the signal , noise and chosen interference
    Received =(Int->add(Codedsig)).add(GN);

    //  Final will store the an estimation of the original signal at the receiver
    Signal<double> final(B);

    //  Make decisions based on the signal and thus attain the final estimated signal
    final=Receiver(Received,chips,64,B,1000.0);

    // Calculate the difference between the transmitted and received signal
    long cnt=Transsig.NoofErrors(final);

    cout<<"The result for SNR "<<SNR<<"db(s) is "<<cnt<<" Errors"<<endl;//Result of the simulation

    return 0;
}
```

## *MISSION OF AFRL/INFORMATION DIRECTORATE (IF)*

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.